PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Utilisation d'analyse de concepts formels pour la gestion de variabilité d'un logiciel configuré dynamiquement
Auteur: Author:	Théotime Menguy
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Menguy, T. (2014). Utilisation d'analyse de concepts formels pour la gestion de variabilité d'un logiciel configuré dynamiquement [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1439/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1439/
Directeurs de recherche: Advisors:	Merlo Ettore
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

UTILISATION D'ANALYSE DE CONCEPTS FORMELS POUR LA GESTION DE VARIABILITÉ D'UN LOGICIEL CONFIGURÉ DYNAMIQUEMENT

THÉOTIME MENGUY
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUIN 2014

© Théotime Menguy, 2014.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

UTILISATION D'ANALYSE DE CONCEPTS FORMELS POUR LA GESTION DE VARIABILITÉ D'UN LOGICIEL CONFIGURÉ DYNAMIQUEMENT

présenté par : MENGUY Théotime
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GAGNON Michel, Ph.D., président
M. MERLO Ettore, Ph.D., membre et directeur de recherche
M. KHOMH Foutse, Ph.D., membre

Aux poseurs de questions,
pour ne pas donner les réponses...

REMERCIEMENTS

Au terme de cette aventure, je tiens tout d'abord à remercier mon directeur de recherche, Ettore Merlo, pour son encadrement, ses conseils et pour m'avoir permis de mettre un pied dans le monde de la recherche. Un grand merci également aux partenaires indutriels et institutionnels du CRIAQ, notamment CMC Electronique et plus précisément Martin Gagnon et Neset Sozen pour l'apport de leur expertise industrielle et pour l'éclairage fourni sur le système ayant permis d'interpréter les résultats.

Il me semble également indispensable de remercier mes camarades de laboratoire François, Thierry, Marc-André et plus récemment Mathieu pour leur accueil, leurs conseils et leur aide.

Merci également à ma famille, qui, quoique loin, ne m'a jamais vraiment quitté et à mes colocataires Philippe et Laurent pour leur support et leur patience au cours des deux dernières années.

RÉSUMÉ

L'industrie avionique, extrêmement critique, se trouve également extrêmement contrainte ; par les normes de sécurité et de certification d'une part, mais aussi par les besoins de personnalisation de ses clients d'autre part. Dans ce contexte, la gestion de variabilité des systèmes est un problème de fond des projets de ré-ingénierie de systèmes avioniques. Nous présentons dans ce mémoire des travaux visant à aider la gestion de variabilité en s'appuyant sur l'analyse de concepts formels et sur le web sémantique.

Le premier objectif de recherche consiste à identifier des comportements typiques et des interactions pour les variables de configuration d'un logiciel configuré dynamiquement. Pour identifier de tels éléments, nous nous sommes servi de l'analyse de concepts formels à différents niveaux de précision dans le système ainsi que de la définition de nouvelles métriques sur le système. Pour répondre à ce premier objectif nous avons défini une typologie des variables de configuration et de leurs interactions. Nous avons également étudié les partages de contrôles entre variables de configuration au niveau du code.

Un autre objectif de recherche était de construire une base de connaissance permettant de recenser les résultats des différentes analyses effectuées, mais aussi d'ajouter tout nouvel élément pouvant aider à la gestion de variabilité, notamment à la définition des processus de ré-ingénierie pour chacune des catégories de la typologie. Pour répondre à cet objectif, nous avons construit une solution fondée sur le web sémantique, en définissant une nouvelle ontologie de description, extensible, et permettant la construction d'inférence pour les traitements évoqués plus haut.

Les travaux présentés ici représentent, à notre connaissance la première typologie de variables de configuration pour un logiciel configuré dynamiquement, mais aussi l'application au domaine de l'aéronautique des techniques de documentation et de gestion de variabilités basées sur le web sémantique. Les travaux effectués et les résultats montrent que l'analyse de concepts formels permet effectivement de comprendre certaines propriétés et interactions des variables et que le web sémantique fournit les outils adéquats pour conserver et exploiter les résultats. Toutefois, l'utilisation de l'analyse de concepts formels à partir d'autres relations booléennes, telles que l'appartenance d'une variable de configuration à un produit, et la construction de nouvelles inférences plus précises permettraient de tirer de nouvelles conclusions. L'application de la méthode à d'autres systèmes permettrait également de valider la pertinence de la classification dans d'autres contextes.

ABSTRACT

Because of its critical nature, avionic industry is bound with numerous constraints such as security standards and certifications while having to fulfill the clients' desires for personalization. In this context, variability management is a very important issue for re-engineering projects of avionic softwares. In this thesis, we propose a new approach, based on formal concept analysis and semantic web, to support variability management.

The first goal of this research is to identify characteristic behaviors and interactions of configuration variables in a dynamically configured system. To identify such elements, we used formal concept analysis on different levels of abstractions in the system and defined new metrics. Then, we built a classification for the configuration variables and their relations in order to enable a quick identification of a variable's behavior in the system. This classification could help finding a systematic approach to process variables during a re-engineering operation, depending on their category. To have a better understanding of the system, we also studied the shared controls of code between configuration variables.

A second objective of this research is to build a knowledge platform to gather the results of all the analysis performed, and to store any additional element relevant in the variability management context, for instance new results helping define re-engineering process for each of the categories. To address this goal, we built a solution based on a semantic web, defining a new ontology, very extensive and enabling to build inferences related to the evolution processes.

The approach presented here is, to the best of our knowledge, the first classification of configuration variables of a dynamically configured software and an original use of documentation and variability management techniques using semantic web in the aeronautic field. The analysis performed and the final results show that formal concept analysis is a way to identify specific properties and behaviors and that semantic web is a good solution to store and explore the results. However, the use of formal concept analysis with new boolean relations, such as the link between configuration variables and files, and the definition of new inferences may be a way to draw better conclusions. The use of the same methodology with other systems would enable to validate the approach in other contexts.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ANTLR	*ANother Tool for Language Recognition*
AST	Arbre syntaxique abstrait (*Abstract Syntax Tree*)
CFG	Graphe de flux de contrôle (*Control Flow Graph*)
CRIAQ	Consortium de Recherche et d'Innovation en Aéronautique au Québec
FCA	Analyse de concepts formels (*Formal Concept Analysis*)
FMS	Système de gestion de vol (*Flight Management System*)
FODA	*Feature Oriented Domain Analysis*
LOC	Ligne de code (*Line Of Code*)
MDD	Développement dirigé par les modèles (*Model Driven Development*)
RDF	Modèle de description de ressources (*Ressource Description Framework*)
SPARQL	Protocole et langage d'interrogation de données RDF (*SPARQL Protocol And RDF Query Language*)
SPL	Lignes de produits logiciels (*Software Product Lines*)
UML	*Unified Modeling Language*
XML	*Extensible Modeling Language*

CHAPITRE 1

INTRODUCTION

Les systèmes avioniques sont extrêmement critiques en raison des dégâts qu'ils peuvent causer, en termes de vies dans les pires cas ou plus simplement de coûts matériels. C'est pour cette raison qu'ils sont soumis à de très fortes contraintes au niveau du respect des standards et des normes de qualité et que les industries sont tenues d'obtenir et de conserver certaines certifications.On peut par exemple penser au critère de couverture de test MC/DC [1] ou à la certification de processus CMMI [2]. Bien entendu, ces contraintes impliquent de nombreux surcoûts, tant pour appliquer les normes que pour obtenir les certifications. Afin de limiter les surcoûts, le cycle de vie des systèmes avioniques est extrêmement rallongé et les techniques de développement conservées plutôt que de risquer de perdre la certification en les faisant évoluer. C'est dans ce contexte qu'est né le projet NextGen [3], porté par les compagnies d'Amérique du Nord, qui a pour but de moderniser le développement des systèmes avioniques d'ici 2025 afin de le faire correspondre aux nouvelles normes. L'objectif du projet est de faire évoluer les systèmes actuels, de manière à conserver les certifications en place tout en réduisant les coûts de certification et la facilité d'adaptation à de nouvelles normes.

Afin de moderniser leur fonctionnement et leurs produits, les compagnies avioniques sont intéressées par la méthode de développement dirigée par les modèles (Model Driven Development, MDD) [4]. Ce choix s'explique par le fait que les modèles sont une représentation synthétique du logiciel permettant d'améliorer la traçabilité au sein du système, de générer automatiquement du code et des tests au lieu de développer à la main et donc, par là-même, de réduire les coûts par rapport aux méthodes de développement actuelles. Toutefois, l'application de cette méthode pose plusieurs problèmes ; tout d'abord, en terme de coût, produire un nouveau système fondé sur cette méthode de développement à partir de zéro représenterait un investissement faramineux qu'aucune compagnie ne peut se permettre sans cesser son activité présente, ce qui est encore plus impensable ; ensuite, repartir de zéro impliquerait de faire entièrement certifier les nouveaux produits alors que les anciens ont cette certification depuis de nombreuses années. Une solution intermédiaire consiste donc à faire la ré-ingénierie des systèmes existants afin de conserver des anciens systèmes tout ce qui peut l'être tout en peuplant les modèles nécessaires à l'application des techniques de MDD. C'est dans ce contexte que s'inscrit ce projet de recherche, dont la finalité est la ré-ingénierie d'un système de FMS (système de gestion de vol, *Flight Management System*).

Afin de pouvoir appliquer au mieux le MDD pour les systèmes avioniques, nous avons considéré une approche par lignes de produits logicielles (Software Product Lines, SPL) [5]. Cette approche nous a semblé plus pertinente car elle est prévue pour le type de produits proposés par les compagnies avioniques, à savoir un même système de base qui a été adapté en fonction des besoins des différents clients. Il est cependant nécessaire de comprendre que la gestion des variabilités et la construction d'un modèle de SPL, des problèmes loin d'être simples à l'origine, se compliquent encore si l'on souhaite conserver les caractéristiques d'un système existant. Une étude de la littérature présentée ici permet de mieux comprendre cette problématique ainsi que les solutions envisageables. Nous proposons ici une méthodologie permettant, une fois identifié le code correspondant aux différentes fonctionnalités, de déterminer les caractéristiques des variables de configuration par rapport au contrôle qu'elles exercent au sein du code et ce afin d'améliorer la compréhension du système et de faciliter son évolution. Nous proposons également une étude des interactions entre variables au travers du code qu'elles contrôle et enfin nous présentons la base de données dont nous nous sommes servis pour obtenir et conserver les résultats de nos analyses.

1.1 Définitions et concepts de base

1.1.1 Lignes de produits logiciels

Les lignes de produits logicielles représentent une solution intéressante lorsque l'objectif est de mettre au point un produit disponible en plusieurs versions selon les désirs et besoins des clients. C'est exactement le cas de figure auquel sont confrontés les constructeurs avioniques, leur objectif étant de proposer un produit particulier, qui puisse cependant être adapté à chaque avion selon les besoins. Dans les logiciels actuels, l'entreprise a pu opter pour la production de plusieurs logiciels très similaires mais distincts, ou bien pour un logiciel configuré dynamiquement et pour lequel le choix de configuration permet les variations. La variabilité du système est alors au cœur du système permettant d'adapter le logiciels à un avion spécifique.

Les modèles de SPL reposent le plus souvent sur deux notions de base permettant d'identifier ce qui appartient à une version ou à une autre. Ces deux principes sont :

- **Les éléments communs** : Les éléments communs sont ceux que l'on retrouve quelle que soit la version du logiciel considérée.
- **Les variabilités** : Au contraire, les variabilités ne font pas partie de tous les logiciels de la ligne de produits, elle peuvent être spécifiques à l'un d'entre eux ou bien commune à plusieurs. Certaines distinctions sont présentées plus bas.

On peut dénombrer de nombreux langages de modélisation pour les SPL, plusieurs d'entre

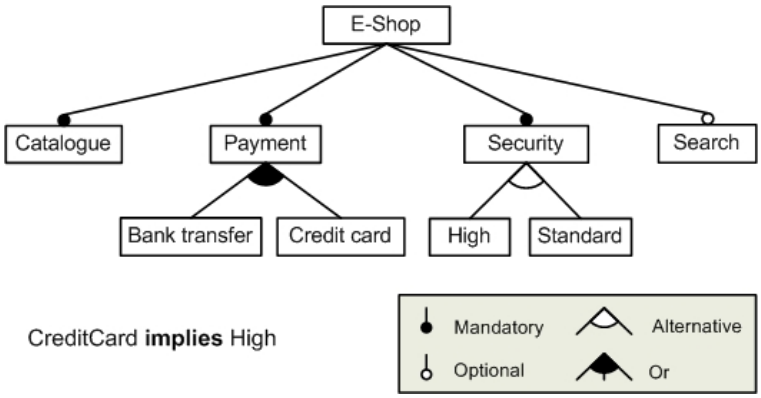

Figure 1.1 Exemple de modèle de fonctionnalités utilisant la représentation FODA pour un distributeur automatique

eux sont présentés dans le second chapitre du présent mémoire, cependant, nous utiliserons ici principalement le langage FODA (*Feature Oriented Domain Analysis* [6]), qui nous semble plus adapté pour certains besoins. La majorité des modélisations regroupent cependant les concepts suivants du fait de leur présence quasi-systématiques dans les systèmes étudiés :

- **Indispensable** : Un tel composant doit nécessairement faire partie de tous les produits de la lignes de produits sans aucune modification.
- **Variation** : Ce type de composants est au coeur de la notion de variabilité, en effet il s'agit des composants dont il existe plusieurs versions et l'une de ces versions est indispensable à tous les produits.
- **Option** : Les options sont moins critiques que les variations car elle ne sont pas indispensables au système, elles sont simplement ajoutées lorsque les exigences des clients le nécessitent, mais certains produits ne les possèdent pas.

Une fois le modèle de SPL construit, et quelle que soit la modélisation choisie, il est possible de générer le produit en sélectionnant les composants que l'on souhaite pour le produit, c'est à dire les variations pertinentes et les options voulues. Les techniques de MDD aboutissant normalement à une génération complète du code une fois tous les composants déterminés. La figure 1.1, provenant de [7], présente un exemple de modèle de fonctionnalités pour un distributeur automatique. On peut bien voir qu'il est possible de choisir les composants à intégrer dans le système pour aboutir à des produits différenciés par leurs options et alternatives ("alternative" est exclusif tandis que "or" ne l'est pas).

1.1.2 Développement dirigé par les modèles

Dans le contexte du développement de logiciel, plusieurs approches sont possibles, l'une des principales de nos jours est le développement dirigé par les modèles ou MDD, comme le

mentionnent MacDonald *et al.*[4]. Cette méthode de développement met en avant les artefacts de modélisation plutôt que le code lui-même, la conception du logiciel est donc au cœur de cette méthode et doit pouvoir définir les solutions à plusieurs niveaux. Lorsque la phase de conception est terminée, dans un contexte idéal, il devrait être possible de générer la quasi-totalité du code automatiquement, avec quelques ajouts de code écrit manuellement pour compléter les quelques lacunes des modèles. De plus, si les modèles sont complets, certaines applications des méthodes formelles permettent la génération d'une partie des tests de manière automatique.

Le MDD présente donc de nombreux avantages puisqu'il permet de raisonner à un niveau d'abstraction élevé et donc impose et permet à la fois une excellente compréhension du produit final, puisque celui-ci correspond a priori en tout point au modèle. Cela suppose bien sûr que l'on considère le générateur de code comme sans faille et que les différents modèles représentent la totalité du système. C'est à ce niveau qu'apparaissent les défauts du MDD. En effet, les langages de modélisation sont souvent moins flexibles que le code lui-même et il peut donc être difficile voire impossible de représenter l'implémentation de certaines parties de la solution. De plus, modéliser de grands changements dans un système pourra passer plus facilement par une nouvelle modélisation que par des modifications d'un modèle existant. Dans le contexte qui est le nôtre, un avantage remarquable du MDD reste cependant la forte traçabilité qu'il permet, qui facilite donc à la fois la compréhension du système et sa certification.

1.1.3 Analyse de concepts formels (*FCA*)

L'analyse de concepts formels est une analyse proposée en 1999 par Ganter et Wille dans [8]. Fondée sur l'utilisation de treillis de Galois, elle permet d'identifier des relations entre différents ensembles d'éléments. Les treillis de Galois sont des treillis pouvant être construits à partir des rectangles maximaux d'une relation binaire, c'est à dire les sous-ensembles maximaux partageant la relation binaire.La FCA construit un treillis de concepts à partir d'une matrice, ou contexte formel, représentant des relations binaires entre deux ensembles d'éléments, les *objets* et les *attributs*, ou respectivement *extensions* et *intensions*. Nous définissons ici les éléments de base nécessaires à la compréhension de cette analyse.

Un contexte formel est un triplet (G, M, I) tel que G et M sont respectivement les ensembles d'*extensions* et *intensions*. $I \subseteq G \times M$ est une relation binaire entre G et M.

Dans le contexte (G, M, I), un concept formel est alors un doublet (X, Y) tel que $X \subseteq G$ et $Y \subseteq M$ et vérifiant les propriétés suivantes :

- $X = \{g \in G \mid (\forall m \in Y)\ gIm\}$, ce qui signifie que X est un ensemble d'*extensions* partageant tous les *attributs* dans Y.

– $Y = \{m \in M \mid (\forall g \in X)\ gIm\}$, soit que Y est un ensemble d'*intensions* partagées par tous les *objets* dans X.

Le treillis de Galois résultant de la FCA est alors partiellement ordonné selon la relation d'ordre partiel suivante : $(X_1, Y_1) \preceq (X_2, Y_2)$ si $X_1 \subseteq X_2$ ou alors si $Y_1 \supseteq Y_2$.

On dit alors qu'un concept c est étiqueté avec un *objet* $g \in G$ si c est le plus petit concept dans lequel g apparait. Inversement, un concept c est étiqueté avec l'*attribut* $m \in M$ si c est le plus grand concept contenant m.

La figure 1.2, tirée de [9], présente le treillis de concept entre les nombres de 1 à 10 et les propriétés pair (e), impair (o), divisible (c), premier (p) et carré parfait (s). Comme on peut le voir, on peut déduire le contenu de chaque concept à l'aide des opérations d'union et d'intersection sur les intensions et extensions des concepts parents ou enfant dans le treillis.

1.2 Éléments de la problématique

Dans le contexte du domaine de l'avionique présenté plus haut, plusieurs compagnies souhaitent moderniser leurs méthodes de développement pour passer à un mode de développement dirigé par les modèles. Étant donné les coûts engagés dans les produits existants, tant en termes de développement (certains logiciels sont en développement depuis plus d'une dizaine d'années), qu'en termes de certification, ces entreprises souhaitent conserver autant que faire se peut le code existant et les algorithmes critiques des différentes fonctionnalités. L'une des contraintes du présent projet est donc de conserver autant que possible le code certifié tout en opérant la transition vers le MDD, ce qui nécessite un traitement automatique des données. En effet les systèmes concernés sont de taille trop importante (plusieurs centaines de milliers de lignes de code) pour envisager un traitement manuel du problème. L'identification du code relié aux différentes fonctionnalités a été effectuée lors d'une précédente étape du projet et il s'agit donc désormais de traiter les données extraites du système pour proposer une méthode de ré-ingénierie respectant les différentes contraintes. La méthodologie proposée ici vise donc d'une part à aider au rassemblement des fonctionnalités en ligne de produits en identifiant des comportement typiques qu'il serait possible de réunir.

La notion de lignes de produits est intrinsèquement liée au domaine de l'avionique et ce pour plusieurs raisons. Tout d'abord, la plupart des compagnies se spécialisent sur un ou plusieurs logiciels nécessaires à un appareil et tentent de le proposer à différentes compagnies de constructeurs. Chaque logiciel doit donc être adaptable en fonction des spécifications de chaque client. Par ailleurs, pour chacun des constructeurs il doit également être possible de personnaliser le logiciel pour chacun des types d'appareils sur lesquels il a vocation à être installé. Cette personnalisation inclut l'adaptation à différents modèles d'avions ou d'hélico-

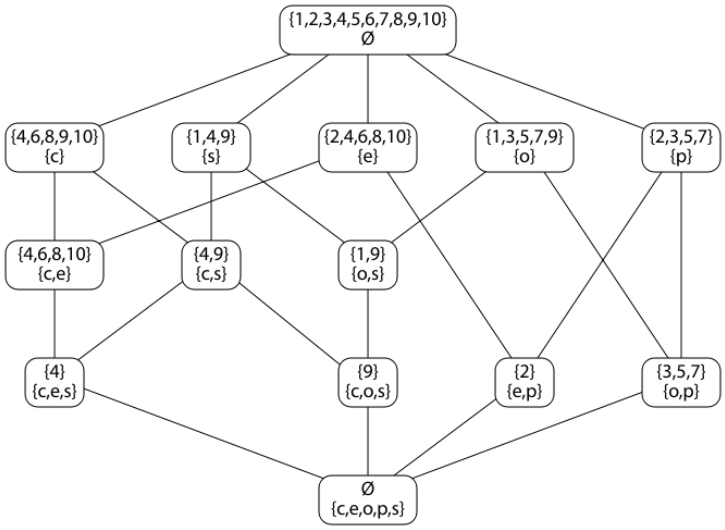

Figure 1.2 Exemple de treillis issu de l'analyse de concepts formels

ptères mais peut aussi dépendre d'attentes spécifiques des clients du constructeur, une compagnie aérienne locale n'attendant pas nécessairement tout à fait le même produit qu'une compagnie transatlantique par exemple. Un modèle de SPL pourrait donc être une méthode appropriée pour implémenter le MDD sur des systèmes avioniques. Un modèle SPL rassemble les fonctionnalités selon leurs affectations aux différentes versions du logiciel de manière à regrouper les fonctionnalités attribuées aux mêmes versions et à séparer celles n'ayant rien en commun. Au cours de cette recherche, nous avons identifié des comportements de variables de configuration susceptibles d'aider la prise de décision pour la refactorisation du code et le découplage entre fonctionnalités pour faciliter l'extraction de lignes de produit.

Lorsque l'on considère l'extraction de lignes de produits et, de manière plus générale, la gestion des variabilités dans un système, l'utilisation de la FCA semble, comme le confirme la revue de littérature du chapitre suivant, tout à fait appropriée à plusieurs niveaux. Dans un premier temps, on peut noter qu'il s'agit d'une analyse automatisable, ce qui est tout à fait appréciable lorsque l'on s'intéresse à des problèmes de grande taille comme ceux qui nous intéressent en avionique. Dans un second temps, l'analyse de concepts formels identifie la totalité des ensembles se partageant *intensions* ou *extensions*, selon l'angle par lequel on le considère, et il est donc possible de s'assurer, lorsque l'on regroupe des fonctionnalités par ce biais, qu'aucun produit n'a été oublié, ni aucune fonctionnalité. La particularité de certains concepts tel que le concept maximum et le concept minimum permet également d'identifier des éléments communs à tous les produits ou au contraire présents dans aucun. La méthodologie proposée ici s'appuie donc sur plusieurs applications de FCA avec différents

ensembles d'intensions et d'extensions, combinés pour en tirer des propriétés.

Ce mémoire présente des travaux de recherches effectués dans le cadre d'un projet du Consortium de Recherche et d'Innovation en Aérospatiale au Québec (CRIAQ). Trois compagnies du domaine de l'avionique collaborent au projet en permettant l'accès à certains de leurs systèmes et en finançant une partie des travaux, il s'agit de CAE inc., CMC Electronique inc. et Mannarino Systems & Software. L'objectif général du projet vise à appliquer le MDD et les méthodes formelles pour développer des logiciels certifiés d'avionique, ce qui inclut la recherche de méthodes pour faire la ré-ingénierie des logiciels existants afin de peupler des modèles de SPL ou plus simplement de MDD selon les cas.

Au sein du projet CRIAQ, les recherches présentées dans ce mémoire correspondent à une collaboration plus étroite avec CMC Électronique, puisque le logiciel étudié est leur système de gestion de vol (*Flight Management System*, FMS) auquel nous avons eu accès. Ce logiciel est composé d'environ cinq cent milles lignes de code en C/C++ et en assembleur et est développé depuis plus de quinze ans. Le cycle de développement actuel du système et sa conception ne permettent pas d'appliquer le MDD directement. Une ré-ingénierie est donc nécessaire et le logiciel résultant doit être modélisé sous forme de SPL afin de respecter les contraintes de variabilité expliquées précédemment. L'objectif sous-jacent et à plus long terme de cette recherche est donc la construction d'un modèle SPL et son peuplement à partir des observations faites dans la présente recherche.

1.3 Objectifs de recherche

Les objectifs de recherche sont les suivants :

- Définir une classification des variables de configuration et de leurs relations au sein du code source d'un logiciel configuré dynamiquement ;
- Analyser les partages de contrôle de code entre variables
- Définir un format de base de données pertinent pour les résultats d'analyses
- Implanter la solution trouvée et valider son intérêt pour l'entreprise partenaire.

1.4 Plan du mémoire

La suite de ce mémoire est composée de cinq parties. Le second chapitre présente une revue critique de la littérature relevant des problématiques reliées aux objectifs de recherches tels que la représentation d'une SPL, la construction d'un modèle de SPL ou son extraction et enfin l'utilisation de la FCA pour gérer les variabilités d'un système.

Le troisième chapitre est consacré aux diverses analyses menées sur le logiciel et aux conclusions qui ont pu en être tirées. Il s'agit notamment de présenter les données disponibles

lors du début des travaux et les choix d'analyses menées ainsi que les motivations de ces choix, mais ce chapitre présente également les productions et conclusions obtenues au terme des analyses.

Le quatrième chapitre s'intéresse à la création de la base de données permettant de rassembler les résultats des analyses, ce qui inclut les contraintes pesant sur la mise en place ainsi que les solutions envisagées et la solution implantée à terme.

Dans le cinquième chapitre, nous présentons l'analyse de certains partages de contrôle entre variables de configuration.

Le sixième chapitre présente les conclusions tirées de ces travaux ainsi que leurs limites et les travaux à envisager dans le futur.

CHAPITRE 2

REVUE DE LITTÉRATURE

Dans cette section sont présentés de manière critique plusieurs pans de la littérature touchant des sujets reliés à la problématique de recherche du présent mémoire. Trois aspects principaux de cette recherche sont abordés dans cette revue de littérature, tout d'abord, nous nous intéressons à la problématique de modélisation et d'extraction de lignes de produits logicielles et aux diverses solutions proposées jusqu'ici pour y répondre. La troisième section présente un certains nombre d'approches fondées sur la FCA afin de gérer et/ou d'extraire les variabilités d'un système pour en faire la maintenance ou la ré-ingénierie. Enfin, la dernière section traite de l'utilisation de technologies du web sémantiques pour la gestion de systèmes, leur analyse ou leur documentation.

2.1 Représentation d'une SPL

La représentation d'une SPL est un problème important dans le cadre de la recherche présentée dans ce mémoire dans la mesure où l'on souhaite faciliter l'extraction des lignes de produits et qu'il sera donc nécessaire de produire des résultats pouvant être représentés dans un modèle de SPL. L'approche MDD visée rend le choix de modélisation d'autant plus critique qu'une représentation inadaptée rendrait l'approche couteuse et inefficace. Cette sous-section s'attache donc à présenter les propositions de la littérature les plus pertinentes dans notre contexte. Néanmoins, des dizaines de modélisations existent et cette présentation ne saurait se faire exhaustive.

Parmi les différentes alternatives, l'une des premières qui nous soit apparue comme pertinentes est la modélisation de variabilités avec UML (*Unified Modeling Language*), proposée par Gomaa [10]. Cette alternative présente l'avantage d'utiliser des représentations et des diagrammes déjà connus, tant dans la communauté académique qu'en industrie. Elle est donc d'une réutilisation facile dans d'autres circonstances et l'adjonction de nouveaux diagrammes si le besoin se fait sentir peut se faire simplement. Cette modélisation présente par ailleurs un intérêt non négligeable par la représentation aisée de classes d'éléments, pouvant faciliter des inférences directes. Néanmoins, si la modélisation UML est connue et facilement lisible pour la plupart de la communauté informatique, c'est également une modélisation dont la représentation est extrêmement lourde lorsque l'on s'en sert pour du MDD, de nombreuses

vues étant nécessaires, ce qui, dans le cas d'un système de grande taille pose des problèmes de lisibilité des modèles.

Une autre approche de modélisation a été proposée par Sinnema *et al.* [11] avec COVAMOF (*COnIPF VAriability MOdeling Framework*). Si cette approche de modélisation conserve les concepts de base tels que *obligatoire, alternatif* ou *optionnel*, la nouveauté se situe dans la conception des critères d'identification d'une variabilité. Les travaux précédents se fondaient sur l'utilisation de critères de dépendance précis quoique souvent adaptables au contexte, mais néanmoins fixés au début du processus de modélisation. COVAMOF s'extrait de cette contrainte par l'identification de dépendances complexes au travers d'analyses dynamiques pouvant identifier des combinaisons de paramètres de configuration aboutissant à des pertes de performance dénotant donc des dépendances positives ou négatives. Il est à noter que de telles dépendances sont complexes à identifier pour la simple raison qu'elles sont interdépendantes, mais il est également très important de les identifier. COVAMOF propose donc une représentation de cet aspect des variabilités. Les limites de COVAMOF se situent d'une part dans la complexité des représentations, les diagrammes étant à la fois nombreux et chargés, la lecture en est plus difficile que celle d'autres modélisation. De plus, la modélisation COVAMOF requiert de nombreuses annotations qui sont difficilement utilisables sur un système de grande taille.

Dans ses travaux de thèse [12], Bak a préféré utiliser le langage de modélisation CLAFER proposé en 2013 [13]. CLAFER est dédié à la représentation de fonctionnalités en tenant compte de leur variabilité et a pour objectif de permettre la représentation de dépendances et de relations qui n'étaient alors pas disponibles au sein des représentations de variabilité existantes. Il est par exemple possible de modéliser un produit particulier en supprimant tous les aspects de variabilités non pertinents d'un système. Cependant, les limites de CLAFER se trouvent être similaires à celles de COVAMOF, c'est à dire que la richesse de représentation du langage nuit à la compréhensibilité du modèle dès lors que le système est complexe. De plus, CLAFER vise à représenter les fonctionnalités et il serait nécessaire d'y faire des adjonctions pour représenter des liens avec du code dans le cadre d'un projet comme celui du présent mémoire.

La modélisation qui a été privilégiée comme référence au cours de notre cas d'étude est la méthode FODA (*Feature-Oriented Domain Analysis*) [6]. Cette modélisation proposée par Kang *et al.* en 1990 se fonde sur une représentation arborescente du système. Chaque sous-arbre représente une portion du système et la forme des arêtes différencie les différentes relations possibles (obligatoire, alternative ou optionnelle). De plus, il est possible de modéliser certaines formes de dépendances par des spécifications supplémentaires.Par exemple, lorsque deux variantes de composants doivent toujours être utilisées ensemble, il est possible

de le spécifier au sein du modèle au travers d'une contrainte. Un exemple de modélisation est présenté à la figure 1.1. La modélisation FODA présente l'intérêt d'être simple et lisible quelle que soit la taille ou la complexité du système puisque sa nature arborescente permet de pouvoir en comprendre une portion sans avoir à saisir la mesure de tout le reste. La simplicité de représentation apporte également la possibilité d'ajouter des éléments de représentation en fonction des besoins et ce, de manière assez simple. La structure de cette représentation présente également l'intérêt de permettre assez simplement tout ajout de composant, de variation ou d'option sans avoir à retoucher la totalité du modèle, ce qui n'est pas si courant en modélisation. Il faut néanmoins noter que c'est précisément la simplicité de FODA qui en fait les limitations, ainsi, les dépendances entre éléments ont une représentation séparée de l'arbre de représentation, choix qui facilite la lisibilité, mais qui nuit à une compréhension visuelle du système, de plus il peut-être complexe de représenter certaines formes de relations entre éléments dès lors qu'elles ne respectent pas les frontières entre sous-arbres. Étant donné la simplicité d'enrichissement de la représentation dans FODA, il n'est pas surprenant de voir que la communauté a proposé de nombreuses extensions telles que CBFM [14] ou CVL [15] permettant de répondre plus facilement à certains besoins de représentation.

Les techniques de modélisation présentées ci-dessus ne représentent bien entendu qu'une partie de ce qui a été proposé par la communauté pour répondre au besoin de modélisation de variabilité. On pourrait également d'intéresser aux travaux de Czarnecki *et al.*[16] qui ont pris un angle d'approche légèrement différent du notre. En effet, ils ont cherché une représentation textuellement complète,au détriment d'une visualisation plus claire que nous voudrions conserver autant que possible. D'autres approches existent telles que VSL [17], PLUSS [18], OVM [5], la technique de Van der Hoek [19] ou encore Feature Assembly [20]. Bien entendu, plusieurs compagnies proposent des systèmes d'extractions avec leur propres représentations, c'est notamment le cas de ConIFP [21] ou GEARS [22]. Dans tous les cas, malgré les approches distinctes et les différents choix faits, toutes ces modélisations se fondent sur les principes de base des lignes de produits présentés dans l'introduction de ce mémoire.

2.2 Extraction de SPL

L'extraction d'un modèle SPL peut se faire dans deux types de contextes distincts : tout d'abord, il est possible, comme c'est le cas pour le présent projet, de souhaiter faire la réingénierie d'un système pour des raisons de coût, d'organisation ou de modernisation. Le point de départ étant alors un logiciel existant pour lequel il faut proposer une nouvelle structure ; le second contexte est celui d'un système déjà existant sous forme de lignes de

produits mais dont la traçabilité est incomplète ou pour lequel certaines connaissances se sont perdues faute de documentation appropriée lors des évolutions du système. Dans les deux cas, différentes méthodologies ont été proposées et nous présentons ici celles qui nous semblent les plus proches de nos travaux.

2.2.1 Réingénierie d'un système existant

L'une des principales difficultés pour extraire un modèle de SPL à partir d'un groupe de produits logiciels similaires est de retracer la correspondance entre une fonctionnalité et le code associé. De nombreuses méthodes ont donc été proposées pour identifier de tels liens et s'en servir pour construire un modèle de SPL. Dans notre cas, nous nous appuyons sur les résultats de [23] pour faire notre étude. Nous présentons toutefois d'autres techniques pertinentes ici car l'application de notre méthodologie à des systèmes qui ne sont pas configurés dynamiquement les nécessiterait.

Une autre difficulté que pose la ré-ingénierie d'un système passe par l'identification des variabilités et l'estimation de la possibilité de réutilisation à partir du logiciel de départ. La communauté scientifique s'est donc penchée sur le problème et le *Software Engineering Institute* propose ainsi les méthode MAP [24] (*Mining Architectures for Product lines*) et OAR [25] (*Option Analysis for Reengineering*) qui sont respectivement dédiées à la conception d'un modèle SPL en analysant les similarités d'une famille logicielle et à l'analyse des possibilités de réutilisation des acquis du système de départ. Malgré l'intérêt évident de ces deux méthodes dans le contexte qui est le notre, elles s'avèrent difficilement applicables de manière systématique et sont peu automatisables en raison du grand nombre de paramètres à déterminer de manière spécifique pour chaque projet. Elles s'avèrent par conséquent difficiles à suivre sur un gros système tel que celui à la base du présent projet, l'objectif final du projet, dont le présent travail n'est que le premier pas, étant d'aboutir à une extraction d'information et une population de modèle automatique, la seule partie manuelle étant la définition d'inférence entre comportement de variables et nouveau modèle.

Alves *et al.* proposent avec FLiP [26] une série d'outils pour l'extraction de lignes de produits à partir de logiciels Java. Leurs outils ont été conçus et testés sur une série de jeux mobiles. Malgré l'intérêt non négligeable de leur approche, elle reste limitée par le champ d'application, les logiciels Java, et par la taille des projets considérés, les jeux mobiles étant par nécessité de taille réduite.

Une autre approche envisageable et proposée par Couto [27] ou encore Zhang [28] s'appuie sur la compilation conditionnelle pour séparer les différents composants les uns des autres. Encore une fois, cette approche, malgré l'intérêt qu'elle présente en termes de facilité de

mise en œuvre et d'automatisation sur de gros systèmes industriels, est limitées aux logiciels conçus à partir de compilation conditionnelle, ce qui restreint non seulement les langages sur lesquels on peut appliquer cette technique, mais plus encore les logiciels puisque la compilation conditionnelle impose de nombreuses contraintes qui ne sont pas toujours conciliables avec le domaine d'application. Ces limites posent donc d'importants problèmes pour une mise en œuvre d'une telle approche.

On peut également noter l'approche de Valente *et al.* [29] utilisant l'outil CIDE pour identifier le code d'une fonctionnalité au travers d'une coloration du code. Cependant, cette approche n'est que partiellement automatisée, ce qui représente un inconvénient non négligeable pour son application sur de gros systèmes, et son utilisation n'a été vérifiée que sur l'extraction d'éléments optionnels de systèmes et la séparation entre éléments nécessaires mais incompatibles n'est pas assurée.

Xue *et al.* [30] à l'inverse proposent une technique automatisée d'analyse de variabilité. En utilisant de la comparaison de modèles, de la détection de clones et certaines techniques d'extraction d'information. Cependant, cette approche pose le problème du besoin de comparaison de produits, ce qu'il est impossible de réaliser dans le cadre d'un système à configuration dynamique. De plus, cette approche nécessite une excellente traçabilité dans le système afin de faire le lien entre l'analyse de modèle et l'analyse du code ; malheureusement, de très nombreux systèmes ne disposent pas de ce genre d'informations avec la précision nécessaire et les limitations de l'approche deviennent alors problématiques.

Comme expliqué dans les paragraphes précédents, les techniques présentées ne sont pas adaptées à nos besoins puisque nous souhaitons traiter un système industriel de grande taille configuré dynamiquement. C'est la raison pour laquelle il a été nécessaire dans [23] de mettre en place une nouvelle approche pour en extraire une ligne de produits répondant aux attentes de nos partenaires industriels. Cependant, le travail n'était pas terminé car les interactions étaient trop complexes pour être analysées simplement, c'est à dire que les interdépendances entre variables se sont avérées plus nombreuses et plus complexes que prévu et qu'il est donc difficile d'identifier une ligne produit à l'aide du seul lien entre variable de configuration et code.

2.2.2 Extraction d'un modèle à partir d'une SPL

L'extraction d'un modèle à partir d'une ligne de produits existante est un problème qui survient souvent lorsque des lacunes sont découvertes au niveau de la traçabilité au sein du système. Il peut également être nécessaire d'extraire le modèle lorsque de gros changements sont envisagés dans le système et que l'on veut garantir la validité des informations dispo-

nibles sur le système.

On peut alors s'intéresser à des méthodes fondées sur la rétro-ingénierie comme c'est le cas pour celle proposée par She *et al.* [31]. Les auteurs de ces travaux ont poussé la recherche d'extraction et de modélisation d'une ligne de produit très loin, puisqu'ils sont capable d'identifier des dépendances de fonctionnalités, des exclusions ou encore des groupes pertinents. Mais leur objectif ne concerne pas une configuration dynamique et se concentre plus sur l'utilisation d'une documentation complète sur les fonctionnalités et leurs dépendances au sein du logiciel. Le lien entre la représentation qu'ils obtiennent et le code est ténue tandis que notre objectif est de conserver autant de code que possible de manière fonctionnelle ; cela s'explique par le fait que les algorithmes avioniques sont le cœur du produit vendu et que les industries ne souhaitent pas devoir les développer à nouveau.

Stoiber *et al.* [32] ont ainsi proposé une approche de tri de fonctionnalité à partir des spécifications graphiques d'un système. Leur approche utilise un genre d'analyse par aspects, c'est à dire qu'à partir du type de variabilité auquel une fonctionnalité est soumise, il leur est possible d'identifier les aspects du système touchés et de les extraire ou de les faire évoluer. Elle permet de faire évoluer un système déjà présent sous forme de ligne de produits, ce qui est particulièrement intéressant lorsque l'on souhaite ajouter de nouveaux degrés de variabilité, ou de personnalisation si l'on change de point de vue, sans pour autant devoir remodéliser ni recoder l'ensemble de la fonctionnalité atteinte. C'est donc une approche qui pourra être pertinente dans la suite du projet, lorsque la ligne de produits de notre système sera effectivement développée.

La comparaison entre les produits d'une même ligne se retrouve bien entendu souvent dans les approches d'extraction de tels modèles. C'est à partir de cette idée que Koschke *et al.* proposent une technique de comparaison d'architecture logicielle [33] fondée sur la technique de réflexion de Murphy [34] extrayant une vue statique du système à comparer avec l'architecture théorique du logiciel. L'approche se déroule de la manière suivante : on commence par un produit qui sert de référence puis, par la suite, pour chaque produit, les éléments qui ne sont pas retrouvés sont ajoutés en tant que variantes. Tout le processus doit cependant être validé manuellement, et rend donc la tâche complexe dès que la taille du système devient importante. De plus, la technique n'a été testée que sur un nombre très réduit d'applications et on peut donc s'interroger sur la possibilité de généralisation.

Une autre approche se servant de la comparaison de différents produits a été présentée par Harhurin *et al.* [35]. L'idée est de comparer les diagrammes de services de différents produits de la même famille logicielle afin de trouver les points communs et les services différents entre produits. Les diagrammes de services sont une modélisation graphique de spécification res-

trictive du système, et sont donc par là-même extrêmement formels. Dans ce contexte, si leur comparaison est simple grâce au formalisme, leur extraction lorsqu'ils ne sont pas disponibles est à tout le moins ardue. En effet, cela nécessite une définition formelle de chaque service, ce qui, pour un système de grande taille est impossible à réaliser simplement, sans parler du fait qu'un logiciel complexe avec de nombreuses interactions présentera des diagrammes particulièrement complexes à extraire. Cette limitation rend l'utilisation de la technique presque impossible sur un système industriel n'ayant pas été conçu à partir de diagrammes de services.

Malgré l'intérêt de ces techniques, il est nécessaire de s'en détacher pour notre projet pour la simple raison que le système à la base du projet n'est pas encore sous forme de ligne de produits, que nous n'avons qu'un seul produit et non plusieurs et que ces techniques ne peuvent donc pas encore s'appliquer. Il est néanmoins nécessaire de tenir compte de leur existence, champs d'application et limites étant donné qu'une fois le système modélisé en SPL, l'application de certaines de ces techniques sera pertinente pour le produit et qu'il faut donc le rendre compatible avec elles.

2.3 Variabilité et FCA

Dans la mesure où la finalité de cette recherche est d'aider à proposer une nouvelle modélisation d'un logiciel existant afin de gérer la variabilité d'une nouvelle manière, il semble logique de s'intéresser à ce que propose la communauté pour atteindre ce type d'objectif. De plus, puisque nous avons choisi de baser notre approche sur la FCA, nous avons limité cette section aux travaux s'appuyant sur cette analyse afin de gérer ou d'identifier des variabilités.

Une étude de Tilley *et al.* [36] recense le type de techniques pour lesquels la FCA a été utilisée dans le cadre du génie logiciel. Dans ce recensement de techniques, il est intéressant de constater que ces approchent touchent en grande partie les domaines de la maintenance de logiciel et de récupération d'information, que ce soit au niveau des spécifications, des modèles ou du code ; cependant, dans ces différents domaines, un aspect qui revient régulièrement concerne le recoupement d'information entre différents résultats d'analyses ou d'ensembles d'éléments caractéristiques. Il est intéressant de voir que la gestion de variabilités est une problématique qui n'est apparue qu'après d'autres champs d'application tels que les contrôles d'accès [37] ou la pure maintenance alors que la définition même de l'analyse induit la recherche de similarités et de différences entre ensembles.

Eyal-Salman *et al.* [38] utilisent l'analyse de concepts formels afin d'identifier, dans un ensembles de produits d'une même ligne, les éléments de la ligne de base et les éléments

alternatifs ou optionnels. Dans ce contexte, la FCA sert à retrouver une traçabilité perdue à partir d'informations extraites du système. Si l'utilisation de la FCA semble améliorer les performances de ce type d'approches, cette approche reste néanmoins limitée à une récupération d'informations sur un système déjà organisé en ligne de produits. La recherche d'éléments communs et spécifiques propre à une gestion de variabilité semble pouvoir être bien gérée par l'utilisation de la FCA.

Dans un autre contexte, la reconnaissance des domaines de fonctionnalité d'un système basé sur plusieurs applications peut s'avérer complexe. Pour répondre à ce type de problèmes, Yang *et al.* [39] proposent d'utiliser la FCA à partir des différentes méthodes d'un système et de la sémantique des accès aux données. Différentes méthodes de tri et de traitement sont alors appliquées afin de construire un modèle de domaine de fonctionnalités. Encore une fois, l'analyse de concepts formels est le moyen utilisé afin d'identifier des regroupement pertinents par propriété communes bien que certaines étapes de l'approche restent manuelles et posent donc problème sur des systèmes industriels de grande taille.

Étant donné le caractère extrêmement formel de la FCA, certaines études tentent de répondre à certaines problématiques de recherche en limitant les entrées à différentes modélisations. Cependant, Yang *et al.* [40] proposent, dans leurs travaux, d'utiliser les résultats de cas d'utilisation pour détecter des problèmes d'incompatibilité ou de recouvrement dans les requis et spécifications du système. Cette approche, si elle est efficace pour des traitements à haut niveau d'abstraction est, de l'aveu même des auteurs, peu efficace si l'on cherche à faire de la refactorisation de code ou des traitements à bas niveau.

Comme présenté dans les précédentes sous-sections, l'une des problématiques importantes dans un système à forte variabilité est la traçabilité entre fonctionnalité et code source. Kazato *et al.* [41] se sont alors penchés sur la possibilité d'utiliser la FCA comme méthode pour identifier les liens. Il s'agit d'étudier différentes traces d'exécution censées exécuter ou non les fonctionnalités. La FCA entre alors en jeu pour retrouver les ensembles de code exécutés dans les concepts regroupant des exécutions avec certaines fonctionnalités communes. Cette technique présente l'intérêt d'assurer des recoupements d'exécution, ce que ne permet pas l'analyse statique, néanmoins, l'aspect manuel de la définition des scénarii d'exécution pose de nombreux problèmes pour un système de grande taille dont, qui plus est, les traces d'exécutions possibles sont extrêmement nombreuses.

Toujours concernant la traçabilité, les travaux de Satyananda *et al.* [42] en 2007 proposent d'utiliser une décomposition fonctionnelle du système en éléments caractérisés par des mots clés. La décomposition se fait, d'une part au niveau des fonctionnalités et, d'autre part, au niveau des composants architecturaux. Les concepts du treillis étant construits à partir des similarités dans les décompositions fonctionnelles. Toutefois, cette approche requiert de

fortes manipulations humaines pour construire les décompositions fonctionnelles, d'autant plus lorsque le système impose de nombreuses interactions et beaucoup de petites variations d'une même fonctionnalité. Par conséquent, cette technique s'avère d'un usage très limité en dehors de petits systèmes.

Parmi les travaux utilisant la FCA pour la gestion de variabilité, on peut remarquer ceux de Loesch *et al.* [43], qui se rapprochent particulièrement de ceux effectués au cours de la présente recherche. En effet, les auteurs ont cherché à identifier des catégories de relations entre fonctionnalités et à proposer un traitement systématique de ces catégories. Les catégories proposées permettent d'identifier les fonctionnalités devant appartenir au cœur de l'application car toujours utilisées, celle devant être retirées de l'application car jamais utilisées, mais aussi les fonctionnalités allant toujours de pair et celles incompatibles entre elles. Dans ce cadre, ils proposent alors différentes techniques pour refactoriser le code en amalgamant les fonctionnalités inséparables et en marquant comme alternatives les fonctionnalités incompatibles.

Les systèmes sur lesquels a été testé cette approche ont prouvé l'utilité de la méthode dans ce contexte. Les auteurs avouent néanmoins que les analyses effectuées, d'une part, ne tiennent pas compte du code de l'application, mais également que la présence d'un trop grand nombre de fonctionnalités implique un treillis de concepts trop complexe pour être analysé comme proposé. C'est l'une des raisons pour lesquelles leurs cas d'étude ont été réduits pour que le nombre de fonctionnalités soit traitable. De plus, les catégories proposées, quoique que théoriquement primordiales, s'avèrent peu pertinentes dans le contexte d'un système patrimonial puisque les redondances y sont évitées et les alternatives de fonctionnalités y sont souvent assez peu nettes.

Les systèmes sur lesquels ont été testé cette approche ont prouvé l'utilité de la méthode dans ce contexte, les auteurs avouent néanmoins que les analyses effectuées, d'une part, ne tiennent pas compte du code de l'application, mais également que la présence d'un trop grand nombre de fonctionnalités implique un treillis de concepts trop complexe pour être analysé comme proposé. C'est l'une des raisons pour lesquelles leurs cas d'étude ont été réduits pour que le nombre de fonctionnalités soit traitable. De plus, les catégories proposées, quoique que théoriquement primordiales, s'avèrent peu pertinentes dans le contexte d'un système patrimonial puisque les redondances y sont évitées et les alternatives de fonctionnalités y sont souvent assez peu nettes.

Loesch *et al.* ont par ailleurs réutilisé leur classification pour une optimisation des variabilités de systèmes [44]. En effet, au delà de l'extraction de lignes de produits, l'utilisation de la FCA permet, au travers de leur méthode de regroupement par concept, de trouver des groupements de fonctionnalités ayant certaines propriétés communes. En effet, en fonc-

tion de la répartition des éléments au sein des concepts, ils en déduisent des propriétés de comportement. Cette approche a été testée sur un système et a permis une réduction de 50% du nombre de variabilités grâce à de pertinentes propositions de refactorisation. Il est donc intéressant de voir que la méthode de classification couplée à des traitements quasi-systématiques est fonctionnelle et peut être particulièrement efficace. Toutefois, les limites mises en évidence au paragraphe précédent persistent et il n'est pas envisageable d'appliquer cette méthode telle quelle dans un contexte de système patrimonial. Ainsi, si certains aspects de la méthodologie sont pertinents pour le présent mémoire, de nombreuses modifications et adaptations sont indispensables pour supporter tout changement de contexte.

En conclusion de cette sous-section, on peut donc remarquer que depuis sa présentation en 1999 par Ganter et Wille [8], la FCA a été utilisée à plusieurs fins en génie logiciel, que ce soit pour de la maintenance ou du traitement d'information extraite. Cependant, si l'on s'intéresse à l'utilisation de la FCA pour la gestion de variabilité, on peut remarquer que la plupart des techniques mises en œuvre se limitent à des traitements de modèles ou de spécifications et ne traite pas d'informations ni d'objets de code directement. De plus, les techniques nécessitent souvent des pré-traitements non automatisés, ce qui interdit leur utilisation sur des systèmes de grandes tailles.

2.4 Représentation d'un système au travers d'une base de données de graphe

L'utilisation du web sémantique en génie logiciel est une pratique qui se développe sous différents aspect. Zhao *et al.* [45] ont d'ailleurs proposé une classification des ontologies en génie logiciel, une ontologie étant la définition des relations et des ressources intégrées au sein d'une base de données de graphe. On peut ainsi considérer dans le désordre et de manière non exhaustive des ontologies pour les processus, pour les spécifications, pour l'architecture ou encore la documentation. Cette classification met bien en évidence l'intérêt que représente le web sémantique comme manière de conserver une base de connaissance appelée à être à la fois interrogée et étendue. Par ailleurs, ces travaux présentent l'utilisation faite des différents types d'ontologie en fonction des phases de développement d'un logiciel et l'on peut constater que le web sémantique semble être particulièrement intéressant pour la maintenance et le contrôle de qualité, mais aussi pour la réutilisation d'artéfacts. Les principales limites de ce types d'approches se situent toutefois dans la complexité de la conception des ontologies et le besoins de spécificité propres des systèmes patrimoniaux, pouvant difficilement réutiliser une ontologie existante.

On peut également remarquer dans la communauté des travaux visant des objectifs proches de ceux de cette étude. On peut ainsi considérer les travaux de Zhai *et al.* [46] visant à la représentation de systèmes flous et complexes. Au delà du type d'application qui peut être vu comme éloigné du présent travail, il faut voir que l'utilisation d'ontologies et de technologies du web sémantique est un moyen de pallier le manque de compréhension d'un système par un être humain devant interagir avec le système, notamment pour le modifier.

Dans le même type de problématiques, Witte *et al.* ont étudié dans leurs travaux [47] l'intérêt d'utiliser un web sémantique pour faciliter les taches de maintenance. Les auteurs proposent la construction d'une ontologie adaptée aux besoins des développeurs en charge de la maintenance d'un système afin de répondre à des questions de traçabilité, de dépendances, d'architecture ou d'autres encore. Ils proposent également une méthode pour peupler automatiquement cette ontologie. Les travaux présentés dans ce mémoire diffèrent de [47] par le type d'informations à intégrer dans la base de données de graphe. En effet, l'objectif ici n'est pas de simplement documenter le système mais plutôt de mettre à disposition des architectes des versions à venir du logiciel, permettant de faire les meilleurs choix de conception par rapport à leurs besoins.

D'autres travaux [48] de Wang *et al.* se concentrent sur la modélisation de fonctionnalités et la vérification de tels modèles de fonctionnalités. L'objectif des leurs travaux était de proposer une représentation ontologique d'un tel modèle ainsi que de proposer un outil pour le développement d'une telle ontologie. La limite de ces travaux par rapport à notre cas d'étude se situe dans le fait que le modèle de fonctionnalités doit préalablement exister, or, dans notre cas, même si une partie du travail a été fait, il n'existe encore aucun lien entre le modèle de fonctionnalités désiré par l'entreprise pour le FMS et le code, et une vérification du modèle est donc un problème qui n'est pas encore posé par le système.

Enfin, dans l'étude [49], les auteurs étudient la possibilité de construire une ontologie en utilisant des relations identifiées dans un treillis de FCA comme base de raisonnement. Leurs travaux montrent que c'est une possibilité intéressante mais ils mettent également en évidence le fait que les analyses FCA, bien que formelles et pouvant assurer des critères précis, permettent également de guider l'intuition pour identifier des relations et des règles pour la construction d'une ontologie. C'est une idée que nous avons exploitée au cours des présents travaux, une partie non négligeable des raisonnements étant basés sur des observations faites sur des treillis FCA.

Nous pouvons donc déduire de l'existence de ces différents types de travaux que la communauté considère les ontologies et le web sémantiques comme une solution pertinente au problème de la passation de connaissances et la construction d'inférences dans le cadre de la maintenance et l'évolution logicielle, y compris sur des systèmes complexes et en s'appuyant

sur d'autres technologies d'analyses.

Nous pouvons donc déduire de l'existence de ces différents types de travaux que la communauté considère les ontologies et web sémantiques comme une solution pertinente au problème de la passation de connaissances et la construction d'inférences dans le cadre de la maintenance et l'évolution logicielle, y compris sur des systèmes complexes et en s'appuyant sur d'autres technologies d'analyses.

Ce chapitre de revue critique de la littérature du domaine nous a permis de constater, d'une part, que le problème posé est un problème que la communauté scientifique a soulevé et pour lequel de nombreuses solutions ont été proposées. Nous avons également pu voir quelles étaient les limites de ces approches et montrer qu'une nouvelle méthode adaptée à des systèmes industriels de grande taille configuré dynamiquement est nécessaire. De plus, nous avons vu que l'analyse de concepts formels est déjà considérée par la communauté comme un outil utile pour la gestion de variabilité d'un système. Cependant, les techniques existantes ne traitent que peu le code directement, se limitant plus au niveau des spécifications, des modèles ou de la sémantique des fonctionnalités. Une nouvelle utilisation de la FCA pour gérer la variabilité en tenant compte du code semble donc une piste intéressante à suivre dans l'extraction de lignes de produits. Enfin, nous avons vu que l'utilisation du web sémantique comme base de connaissances pour décrire un système était une possibilité déjà étudiée par la communauté et que c'était donc une piste valide pour conserver les données issue d'analyses du système à des fins de documentation, de maintenance et d'évolution.

CHAPITRE 3

Analyse du système

L'objectif de ce chapitre est de présenter les différentes analyses qui ont été menées sur le système afin d'en améliorer la compréhension. Le chapitre est donc organisé comme suit : une première partie présente les données disponibles sur le système au début de ces travaux que ce soit des artefacts du système, de la documentation ou des résultats d'analyses menées précédemment, une seconde partie présente les différentes analyses FCA ayant été menées sur le système en incluant les raisons motivant ces analyses et l'utilisation faite des treillis résultants, la section suivante introduit différentes métriques ayant servi à la compréhension du système et enfin la dernière section présente la classification des variables ayant résulté des différentes analyses.

3.1 Données initiales

Avant les travaux présentés dans le présent mémoire, d'autres projets visant à améliorer la compréhension du système avaient été menés, les premiers provenant de l'entreprise, au travers des artefacts du système lui-même tels que le code, les documents de conception ou la liste des configurations en fonction des produits et d'autres au travers d'autres études telles que les travaux de maîtrise de Maxime Ouellet [23] ayant pour objectif de relier les variables de configuration au code contrôlé. Cette section présente donc l'ensemble des données originellement disponibles pour ce projet.

3.1.1 Description du système

Les premières données disponibles pour mener des analyses sont bien entendu des données de description du FMS. Ces données ont toutes été fournies directement par l'entreprise à un stade ou un autre des travaux et ne résultent d'aucune analyse, seulement du développement du système.

Dans la mesure où les présents travaux visent à comprendre le système, il est impensable de ne pas utiliser le code comme donnée de départ. Pour des raisons de confidentialité, le code du FMS n'a pas été fourni directement, mais un accès a été permis au sein des installations de CMC à Montréal. Le code s'articule en 411 fichiers de C/C++ dans une arborescence de répertoires très peu profonde puisque presque tous les fichiers sont dans le répertoire racine du logiciel.

Nous avons également eu à notre disposition les configurations des différents produits. C'est à dire, pour chaque produit, la liste des variables de configuration activées dans ce produit.

Dans un troisième temps, l'entreprise nous a fourni une liste de composants principaux du système. Chaque fichier de code est relié à un seul de ces composants, dépendant de la fonctionnalité implémentée dans le fichier. Le logiciel se divise en dix-huit composants dont la liste est présentée dans le tableau 3.1, chaque composant comportant entre 4 et 85 fichiers. Chaque composant correspond à une fonctionnalité principale au sein du système, il s'agit donc d'une vision haut niveau du logiciel tel qu'il a été pensé à sa conception.

Tableau 3.1 Nombre de fichiers liés à chaque composant

Composants	**Nombre de fichiers liés**
comp1	4
comp2	5
comp3	6
comp4	7
comp5	7
comp6	7
comp7	10
comp8	16
comp9	17
comp10	17
comp11	21
comp12	21
comp13	22
comp14	32
comp15	33
comp16	42
comp17	59
comp18	85

3.1.2 Contrôle des variables sur le code

La seconde partie des données disponibles concerne les données obtenues en identifiant le code contrôlé par chacune des variables de configuration. Outre la relation de contrôle ainsi identifiée, les analyses effectuées ont également produit d'autres artefacts présentés ici.

Maxime Ouellet [23] a réalisé une analyse statique du système pour identifier les dépendances entre le code et la *valeur* des variables de configuration, ce qui va plus loin qu'une

analyse de dépendance entre code et variable. Pour cette analyse, la première étape a été de développer une grammaire du langage C/C++ tenant compte des spécificités et contraintes du domaine de l'avionique. Cette grammaire a ensuite été utilisée pour extraire le graphe de flux de contrôle à l'aide du logiciel ANTLR[50]. Le graphe de flux de contrôle, ou CFG, a alors été annoté afin de marquer les nœuds du graphe pour lesquels une vérification de la valeur d'une variable de configuration était faite. Un automate de vérification de modèle a alors été construit pour parcourir le CFG et y identifier les différents types de contrôles envisageables (voir ci-après). Les résultats sont organisés en quatre fichiers par variable, un par motif de contrôle (voir ci-après), chaque fichier contenant des identifiants pour les lignes de code contrôlées avec ce motif pour la variable considérée.

Comme précisé ci-dessus, les analyses liant les variables de configuration au code contrôlé sont basées sur l'analyse statique du code et le parcours du CFG. Le CFG, une fois annoté avec les informations de contrôle, a donc été enregistré au format XML afin de pouvoir servir pour des analyses ultérieures. Le graphe se compose donc d'éléments XML dont les attributs contiennent un identifiant unique du nœud, le type de nœud et la ligne de code où se trouve le nœud tandis que les arêtes contiennent les identifiants des nœuds de départ et d'arrivée ainsi que les annotations de contrôle qui, elles, sont contenues dans des éléments enfants spécifiant le type de contrôle et la variable considérée. Un exemple des nœuds rencontrés est présenté en figure 3.1.

Au sein du CFG, le format des dépendances peut varier en fonction de deux éléments distincts : tout d'abord, le contrôle peut dépendre de la condition utilisée, il peut s'agir d'une conjonction ou d'une disjonction, le test peut être fait de manière positive sur la valeur ou bien négative en excluant une valeur ; ensuite, la dépendance peut évoluer en fonction de la forme du graphe de flux de contrôle, une conditionnelle simple étant plus simple à tester que des appels inter-procéduraux. On peut alors identifier cinq types de contrôle différents d'une variable sur le code. Le choix d'affectation d'une dépendance sur un motif ou un autre est expliqué précisément dans [23]. Quel que soit le motif p considéré et la variable V, on note $p(V)$ l'ensemble des lignes de code que V contrôle avec le motif p. Les motifs sont alors les suivants :

- **GAIN** : Le motif de contrôle GAIN est utilisé lorsque l'exécution d'une ligne de code est assurée lorsque la variable est activée. Par exemple dans le code de la figure 3.2, la ligne 3 L_3 appartient à $GAIN(var1)$
- **LOSS** : Le motif de contrôle LOSS est l'inverse du motif GAIN, en effet, une variable contrôlant une ligne de code avec ce motif interdit l'exécution du code lorsque la variable est active. Ainsi dans le code de la figure 3.2, la ligne 5 L_5 appartient à $LOSS(var1)$.
- **SOMEHOWPLUS** : Le motif SOMEHOWPLUS apparaît lorsque les conditions d'exé-

```
<node columnBegin="95" columnEnd="100" id="FileNameNode1" lineBegin="77"
      lineEnd="77" truetype="21" type="IF_CONDITION"/>
<node columnBegin="95" columnEnd="100" id="FileNameNode2" lineBegin="77"
     lineEnd="77" truetype="21" type="STATEMENT"/>

<edge source="FileNameNode1" target="FileNameNode2" type="grant">
   <gain>var1</gain>
   <loss>var2</loss>
   <loss>var3</loss>
 </edge>
```

Figure 3.1 Exemple de nœuds du CFG au format XML

cution se compliquent. En effet, l'analyse étant statique, il n'est pas toujours possible d'évaluer une expression logique en tenant compte de toutes les variables impliquées, le motif SOMEHOWPLUS est alors utilisé pour noter que si la variable est active, la ligne de code est plus susceptible d'être exécutée.

- **SOMEHOWMINUS** : Le motif SOMEHOWMINUS est l'inverse du précédent, en effet, sur les conditions complexes, son utilisation dénote que la ligne de code sera peu probablement exécutée si la variable est active.
- **NON CONTRÔLÉ** : La plupart des lignes de code ne sont pas contrôlées par les variables de configuration et il est donc important de noter qu'aucune variable ne contrôle intégralement le code.

La distribution des motifs de contrôle sur le code piloté par les variables de configuration est intéressante. On peut tout d'abord remarquer que le code piloté représente environ un dixième du code du FMS, le reste du code s'exécutant sans dépendre de la configuration. Il

```
1:   //here is a checking condition
2:   if(var1 || !var2){
3:         int x=1;
4:   }else{
5:         int x=2;
6:   }
```

Figure 3.2 Exemple de contrôle d'exécution de code

Tableau 3.2 Proportion du code contrôlé par chaque motif de contrôle

GAIN	LOSS	SOMEHOWPLUS	SOMEHOWMINUS
92.23%	20.79%	9.29%	2.75%

est alors possible d'étudier la répartition des motifs sur le code, présentée dans le tableau 3.2. On peut remarquer que la somme des pourcentages excède 100%. Cela peut facilement être expliqué par les recoupements de contrôle entre les variables. En effet, une ligne de code l peut tout à fait être contrôlée avec le motif GAIN par une variable $V1$ et avec un autre motif par une autre variable $V2$ tel que $l \in GAIN(V1) \cap LOSS(V2)$. Le recoupement entre les deux implique que la ligne de code compte dans les deux pourcentages.
Une autre observation pouvant être faite concerne les proportions elles-mêmes. En effet, le motif de contrôle GAIN est très nettement dominant, et l'on peut donc en déduire que les variables de configuration sont donc beaucoup plus utilisées pour imposer l'exécution de code que pour l'empêcher avec le motif LOSS. De plus, les motifs SOMEHOWPLUS et SOMEHOWMINUS sont remarquablement réduits par rapport aux autres. On peut donc en déduire que l'utilisation de conditions complexes est peu répandue au sein du logiciel, ce qui pourrait potentiellement simplifier un travail de ré-ingénierie.

3.2 Différentes analyses FCA

L'analyse des variabilités au sein du logiciel FMS est complexe pour plusieurs raisons. Tout d'abord, la taille du système rend impossible un traitement à la main de ces variabilités. Le processus serait beaucoup trop couteux pour une industrie et trop long pour un domaine de pointe tel que l'aéronautique. Ensuite, la nature critique du système et le grand nombre de petits détails paramétrisables garantissant le changement de comportement en fonction des caractéristiques de l'appareil empêchent l'utilisation d'inférences simples, le nombre de cas de recoupement possibles étant tout simplement trop grand.

L'analyse de concepts formels, dont l'utilité principale est de trouver des groupements maximaux en fonction de propriétés partagées peut alors permettre d'identifier des types de comportements et des relations systématiques entre éléments. C'est dans cette optique que nous avons décidé d'utiliser la FCA pour analyser les relations entre éléments dans le système. Nous avons donc construit plusieurs treillis FCA à partir des éléments de données fournies et des relations explicites entre eux. Cette section présente les différents treillis construits, leurs intérêts et les conclusions qui ont pu en être tirées.

3.2.1 Variables de configuration-avions

L'un des points centraux dans la gestion de variabilité concerne la définition de lignes de produits pour le FMS, cet intérêt provenant du fait que le FMS correspond en fait à plusieurs produits, la nature du produit dépendant des choix de configuration qui sont faits. Il semblait alors intéressant d'étudier les relations entre les produits et les variables de configuration au cas où l'on pourrait d'ores et déjà identifier des comportement particuliers.

Dans cette analyse, l'ensemble des intensions I est l'ensemble des produits existants, au nombre de 10, et l'ensemble des extensions E est l'ensemble des variables de configuration ; la relation R considérée est la suivante :

$$i \in I, e \in E, iRe \Leftrightarrow e \in configuration(i) \tag{3.1}$$

Le treillis résultant de cette analyse est présenté en figure 3.3 et comporte environ 150 concepts. Les concepts comportant un demi-cercle supérieur plein correspondent aux concepts maximaux d'au moins une extension, ceux comportant un demi-cercle inférieur plein sont les concepts minimaux d'au moins une intension. Ce dernier nombre est particulièrement intéressant si l'on considère le nombre de variables impliquées et la nature combinatoire de possibilités d'association. Malheureusement, le partage de variables entre les produits prend une forme beaucoup trop complexe pour identifier immédiatement des regroupement de variables simples. Toutefois, les implications de Duquenne-Guigues [51], identifiant le fait qu'un élément de l'ensemble des intensions n'est présent dans un concept du treillis que si un autre élément implicant y est également présent, permettent d'identifier une certaine forme de dépendance entre variables puisque la présence de certaines variables est, dans les faits, impliquée par la présence d'une autre au sein du treillis. Par exemple, certaines dépendances de ce type sont connues et voulues au sein du logiciel, mais pas toutes documentées. De plus, on peut identifier certaines variables n'étant jamais utilisées : il suffit de considérer le concept minimal du treillis, les variables qu'il comporte n'étant jamais utilisées dans aucun produit, elles pourraient être obsolètes et donc à éliminer du système.

3.2.2 Produits-code

Si l'on change de point de vue, un produit logiciel peut être vu non comme une configuration mais comme du code source. Le second type d'analyse FCA que nous avons effectué se concentre donc autour des relations entre produits et code source. Dans ce cadre nous avons construit quatre treillis de FCA : chacun rend compte d'un type de relation entre un produit et une ligne de code pilotée. L'objectif était d'identifier de potentiels ensembles de code partagés par de nombreux produits et donc identifiables comme modules dans une ligne

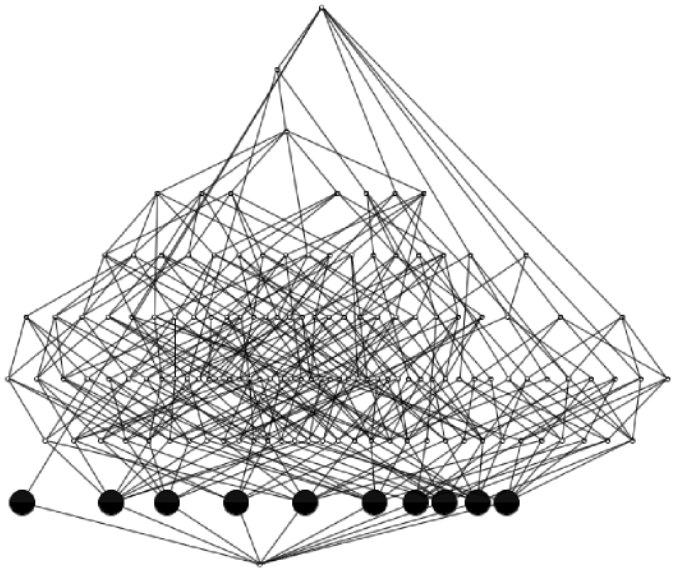

Figure 3.3 Treillis FCA entre variables de configuration et produits

de produits ou au contraire du code spécifique à un produit et donc difficilement intégrable dans un composant partagé.

Dans ces quatre analyses, l'ensemble des intensions I est l'ensemble des produits et l'ensemble des extensions E est l'ensemble des lignes de code pilotées avec le motif considéré selon l'analyse. Pour l'analyse du motif P, la relation entre deux éléments de l'ensemble est la suivante :

$$i \in I, e \in E, var \in configuration(i), iRe \Leftrightarrow e \in P(var) \tag{3.2}$$

Les figures 3.4 à 3.7 représentent les quatre treillis issus des analyses FCA. Une première observation de ces treillis permet de tirer plusieurs conclusions ; tout d'abord, si l'on met en relation la complexité du treillis et la proportion de lignes de code qui y sont impliquées, on peut voir une forme de relation, le treillis semblant se compliquer lorsque le nombre de lignes croît (le nombre de lignes de code dans le treillis du motif GAIN est 45 fois plus important que celui du motif SOMEHOWMINUS). De plus, on peut remarquer que certains produits semblent avoir beaucoup plus d'interactions que d'autres. Par exemple le produit $P9$ semble moins interagir que les autres, quel que soit le motif considéré. On peut donc penser que certains produits se démarquent et pourraient donc être plus facilement séparables lors d'une ré-ingénierie. C'est par exemple le cas du produit $P1$, au sommet du treillis et dont le code semble donc être utilisé de la même manière par tous les avions, il pourrait donc être intégré au coeur de l'application. On peut également voir que le produit $P10$, toujours contenu dans

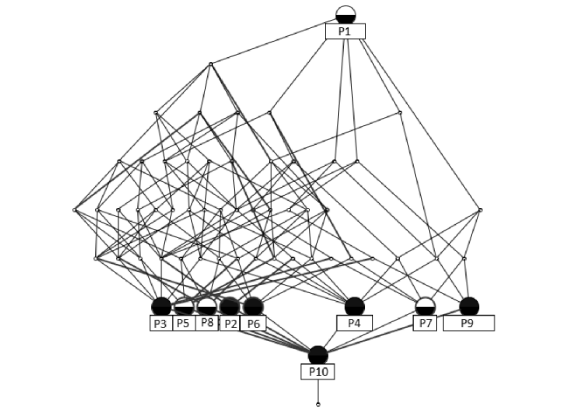

Figure 3.4 Treillis FCA entre produit et code du motif GAIN

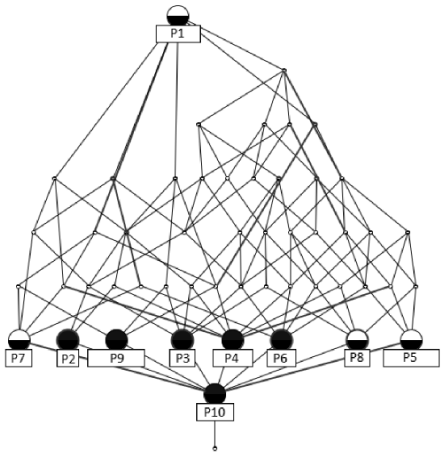

Figure 3.5 Treillis FCA entre produit et code du motif LOSS

un concept inférieur à tous les autres produits et supérieur au minimum contient donc du code exécuté par tous les produits (et devant donc être intégré au coeur applicatif), mais aussi du code spécifique devant être conservé à part.

3.2.3 Variables de configuration-composants

Après avoir étudié les potentiels groupements de variables et de code par rapport aux produits, nous nous sommes posés la question de savoir si l'on pouvait identifier d'autres relations caractéristiques au sein du système. Dans un premier temps notre étude est restée à un haut niveau et nous avons voulu étudier les interactions entre variables par rapports aux composants du système. L'analyse FCA qui a suivi a donc en toute logique été celle fondée sur la relation entre variable et composants affectés.

Dans cette analyse, l'ensemble I des intensions est l'ensemble des composants du système et l'ensemble E des extensions est l'ensemble des variables de configuration. La relation entre les deux correspond au fait qu'une variable contrôle une ligne de code au sein d'un composant et est définie formellement dans l'équation 3.3 avec P l'ensemble des motifs de contrôle et $L(i)$ l'ensemble des lignes de $i \in I$.

$$i \in I, e \in E, iRe \Leftrightarrow \exists p \in P, p(e) \cap L(i) \neq \emptyset \tag{3.3}$$

La figure 3.8 représente le treillis résultant de cette analyse. Malheureusement, si certains comportements sont intéressants, les interactions globales sont difficiles à évaluer vi-

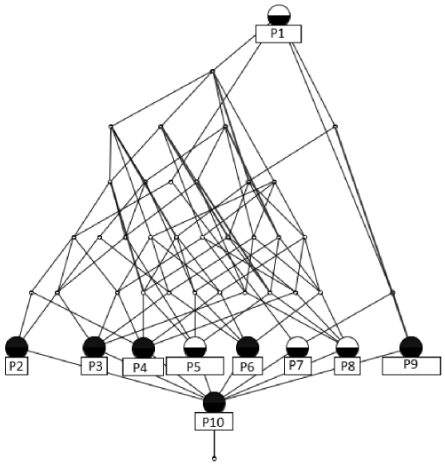

Figure 3.6 Treillis FCA entre produit et code du motif SOMEHOWPLUS

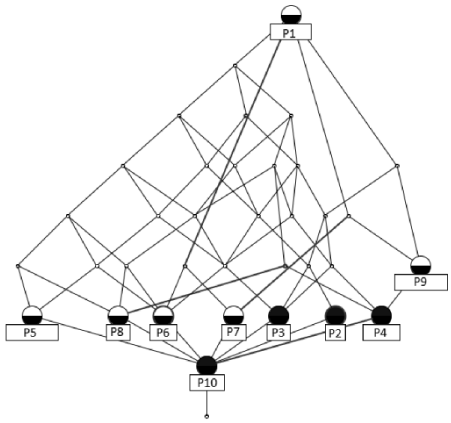

Figure 3.7 Treillis FCA entre produit et code du motif SOMEHOWMINUS

suellement, ce qui est contradictoire avec les attentes de l'industrie partenaire. Parmi les comportements identifiables, on peut déjà remarquer le composant C, dont la position dans le treillis montre que la plupart des variables contrôlant une partie de son code contrôlent également de nombreux autres composants, c'est une interaction globalement ennuyeuse car elle signifie que ce composant est affecté par la plupart des changements pouvant se produire au sein du système. Il faudrait donc envisager de séparer ce composant en modules à rattacher à d'autres composants plus pertinents. À l'inverse, les composants A et B sont relativement isolés dans le système, et pilotés par des variables très spécifiques peu partagées, dans le cadre d'une ré-ingénierie, de tels composants gagneraient à être peu modifiés afin de réduire les coûts de transformation étant donnés qu'ils peuvent déjà être considérés comme des modules fonctionnels.

3.2.4 Variables de configuration-code

Enfin, afin de continuer la recherche d'interactions entre variables au niveau du logiciel mais à plus bas niveau que le treillis précédent, nous avons construit le treillis entre les variables de configuration et les lignes de code contrôlées avec le motif GAIN. Nous nous somme limités à ce motif car c'est le motif le plus représentatif du code exécuté au sein du système. L'objectif de cette analyse est de trouver les variables dont le comportement est plus altéré par d'autres et celles ayant le seul contrôle du code dont elles garantissent l'exécution.

Ce treillis de FCA a été construit en considérant que l'ensemble I des intensions est l'ensemble des variables de configuration et que l'ensemble E des extensions est l'ensemble

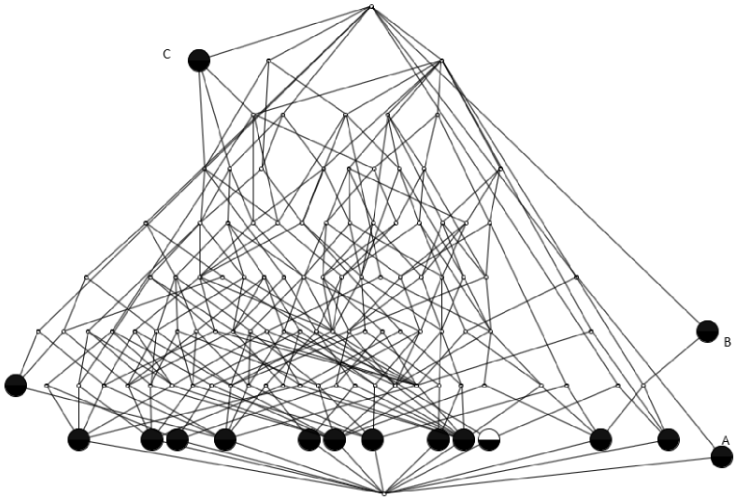

Figure 3.8 Treillis FCA entre variables de configuration et composants

des lignes de code. La relation entre les deux étant la suivante :

$$i \in I, e \in E, iRe \Leftrightarrow e \in GAIN(i) \tag{3.4}$$

En regardant le treillis 3.9 on peut remarquer deux éléments différents au sein du treillis. On trouve, sur la droite de la figure, de nombreuses variables partageant pas ou peu de code avec d'autres variables, l'analyse du code spécifique à ces variables devrait donc être plus facile à faire et à réorganiser. À l'inverse, la gauche du treillis présente des structures de partage extrêmement complexes, et c'est à ce niveau là que doivent se porter les efforts de ré-ingénierie et de maintenance. En effet, les lignes de code contrôlées par trop de variables deviennent difficile à faire évoluer sans risques et les interactions deviennent trop complexes pour que même un expert s'y retrouve. Il serait donc nécessaire d'étudier les possibilités de simplification des variables par des combinaisons de variables lorsque suffisamment de lignes de code sont partagées. Une étude plus approfondie de cette question est proposée au Chapitre 5.

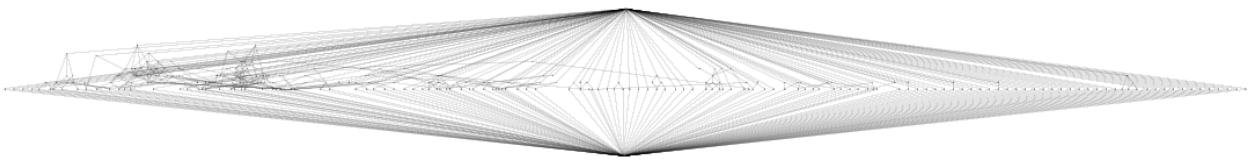

Figure 3.9 Treillis FCA entre variables de configuration et lignes de code

3.3 Nouvelles métriques

Afin de mieux comprendre le système et les caractéristiques des différentes variables de configuration, nous avons considéré qu'il était nécessaire de définir de nouvelles métriques permettant de décrire certains aspects de l'utilisation des variables afin de compléter les informations extraites des différents treillis FCA. Les métriques définies sont calculées à partir des treillis directement ou d'information recoupant plusieurs treillis à la fois. Cette section présente ces métriques, leurs définitions formelles et leurs distributions au sein du système.

3.3.1 Nombre de composants par variable de configuration N_{CV}

L'une des interrogations que l'on peut avoir sur le contrôle qu'une variable exerce sur le système concerne les fonctionnalités auxquelles elle participe. Dans notre cas d'étude, il s'agirait donc de savoir si une variable est dédiée à une fonction en particulier ou bien gère l'interaction entre plusieurs fonctionnalités. Le treillis de la figure 3.8 permet de faire plusieurs observations dans cette direction, mais il est conçu pour identifier des interactions, non des comparaisons. Nous avons donc décidé de définir le nombre de composants par variable, spécifiant le nombre de composants dans lesquels une variable contrôle au moins une ligne de code, quel que soit le motif de contrôle. Ce nombre peut être extrait du treillis variables-composants en récupérant le nombre de composants dans le concept maximal dans lequel est présente la variable. Formellement on peut le définir de la manière suivante :
On considère P l'ensemble des motifs de contrôle, Var une variable de configuration et C l'ensemble des composants du système et pour $c \in C$ on note $L(c)$ les lignes de code affiliées au composant c.

$$N_{CV}(Var) = card(\{c \in C | \exists p \in P, p(Var) \cap L(c) \neq \emptyset\}) \tag{3.5}$$

La figure 3.10 présente la distribution de la métrique pour les variables du système contrôlant effectivement du code, c'est à dire 207 variables. On peut remarquer que plus le nombre de composants augmente, moins il y a de variables réparties sur ce nombre de composants. Par exemple, plus du tiers des variables est limité à un unique composant. Ceci semble confirmer l'idée que si certaines variables sont de haut niveau et contrôlent des éléments complexes du système, la plupart des variables semblent se limiter à un fonctionnalité donnée sur un nombre réduit de composants. Il est également intéressant de voir qu'aucune variable ne s'étend sur plus de 10 composants alors qu'il y a du code contrôlé dans 14 d'entre eux. L'identification des variables de haut niveau est importante dans le cadre d'une ré-ingénierie car ce sont les variables qui devraient subir le plus de modifications.

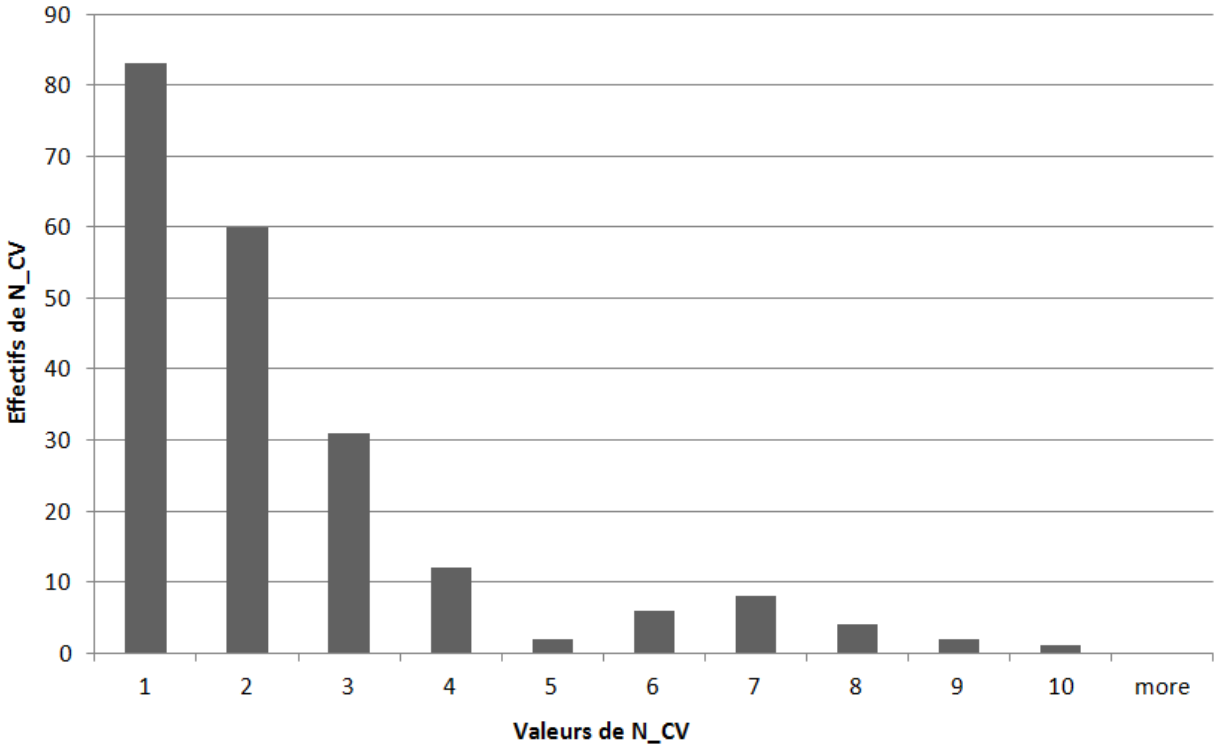

Figure 3.10 Histogramme du nombre de composants par variable de configuration

3.3.2 Nombre de lignes par variable de configuration N_{LV}

Dans un second temps, il peut être intéressant de savoir quel est l'impact d'une variable au sein du code lui même, c'est-à-dire de savoir si la quantité de code piloté par une variable est grande ou non. Ce genre d'observation peut permettre d'identifier les variables de haut niveau, contrôlant de grande quantité de code, et les variables plus proches du détail, réglant des détails de comportements ponctuels. Il nous a alors semblé logique de définir le nombre de lignes de code contrôlées par variable de configuration, indépendamment des composants et des motifs de contrôle. Il s'agit d'associer chaque variable de configuration avec le nombre de lignes qu'elle pilote sans tenir compte du motif de contrôle. La définition formelle se présente alors comme suit :
On considère Var une variable de configuration et P l'ensemble des motifs de contrôle.

$$N_{LV}(Var) = card(\bigcup_{p \in P} p(Var)) \tag{3.6}$$

Comme on peut le voir dans la figure 3.11, les variables de configuration contrôlent des quantités diverses de lignes de code. L'histogramme peut être divisé en trois parties principales. Tout d'abord, les premières 78 variables contrôlent moins de dix lignes de code. Ce sont donc a priori des variables dédiées à un contrôle précis et spécifique au sein du système, non à du contrôle de haut niveau. Ensuite, on peut observer que 82 variables contrôlent entre dix et cent lignes de code, ces variables pouvant avoir des rôles de contrôle moins spécifiques que les précédentes mais restant de la paramétrisation de fonctionnement plus que de la gestion à haut niveau. Enfin, le dernier tiers des variables s'étire jusqu'à près de 30 000 lignes de code pour la variable avec le plus grand contrôle. Cette distribution est intéressante, car la différence entre la moyenne, 379 et la médiane, 26, est lourde de sens. La plupart des variables contrôlent peu de lignes de code mais celles ayant un plus grand contrôle ont une avance suffisante pour augmenter considérablement la moyenne.

3.3.3 Nombre de lignes par composant et par variable de configuration N_{LCV}

Enfin, la dernière métrique de description du comportement des variables de configuration est une sorte de combinaison des deux précédentes. En effet, nous avons envisagé que les variables ayant le moins de lignes de code contrôlé sont présentes dans un unique composant et nous avons souhaité vérifier la possibilité qu'une variable agisse sur plusieurs composants avec un impact différent sur chacun. C'est dans cette optique que l'on définit le nombre de lignes par composant et par variable de configuration pour caractériser la répartition du contrôle des variables sur les composants et identifier des comportements récurrents. Formellement,

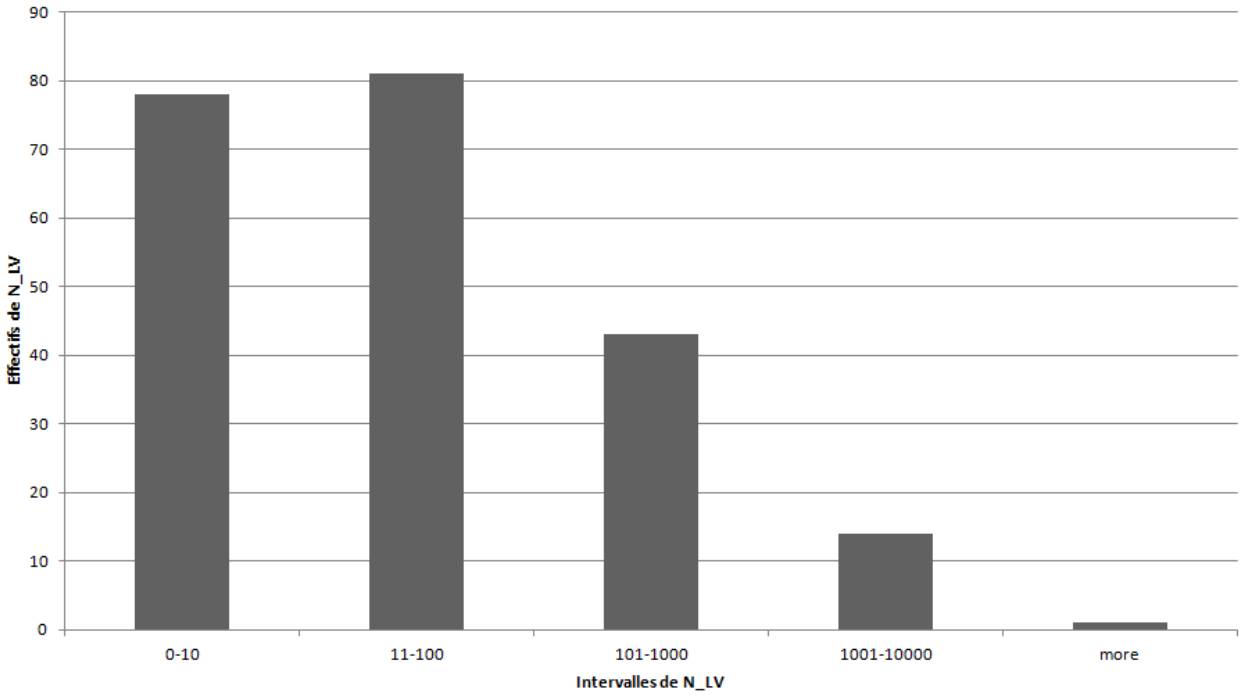

Figure 3.11 Histogramme du nombre de lignes par variable de configuration

la définition de cette métrique prend la forme suivante :
On considère une variable de configuration Var, un composant c du système, l'ensemble P des motifs de contrôle et $L(c)$ l'ensemble des lignes de code affiliées au composant c.

$$N_{LCV}(Var, c) = card((\bigcup_{p \in P} p(Var)) \cap L(c)) \tag{3.7}$$

Nous pouvons remarquer qu'une métrique d'entropie aurait pu fournir une information similaire sur les variables. Cependant, une telle métrique ne permettrait pas de conserver l'information sémantique entre une variables de configuration et les composants sur lesquels elle exerce un contrôle. En effet, une métrique d'entropie rendrait compte de la dispersion d'une variable, mais l'information relative aux fonctionnalités associés serait perdue. Il serait alors nécessaire de la récupérer par la suite, une étape supplémentaire donc, pour pouvoir effectuer la ré-ingénierie appropriée.

Comme on pourrait s'y attendre, la distribution de cette métrique révèle plusieurs organisations typiques. Bien entendu, de nombreuses variables contrôlent du code dans un unique composant. Toutefois, si l'on s'intéresse à la figure 3.12, présentant la répartition du code dans les composants pour les variables contrôlant 3 composants. Chaque colonne de points correspond à une variable, l'ordonnée de chaque point correspondant au pourcentage de code

contrôlé par la variable dans le composant représenté par la forme du point. Il est intéressant de découvrir que certaines variables se répartissent équitablement sur les différents composants du système tandis que d'autres se concentrent sur un composant particulier avec une proportion moindre de code contrôlé dans les autres composants. Nous avons mené cette analyse sur l'ensemble des variables et nous avons découvert que les variables contrôlant le moins de lignes de code sont concentrées dans un unique composant. C'est à dire que toutes les variables contrôlant moins de 26 lignes de code sont concentrées dans un seul composant.

3.4 Une typologie de variables et d'interactions

Afin d'améliorer la compréhension du système et de permettre de définir des traitements quasi-systématiques lors d'une ré-ingénierie du système, nous avons construit une classification des variables de configuration et de leurs relations. L'identification des ces comportements typiques permet également de repérer des anomalies, en effet, un expert du système est capable de savoir, pour une variable de configuration donnée, si le comportement de la variable est optimisé ou non. La typologie permet donc non seulement une meilleure compréhension du système mais aussi de trouver les optimisations de code pertinentes à intégrer. Cette section présente d'abord la typologie des variables, puis les relations induites par les analyses FCA.

3.4.1 Classification des variables

La classification des variables de configuration est un moyen de savoir plus facilement quel est l'impact d'une variable sur le code et sur les autres variables de configuration afin de simplifier les opérations de développement, de maintenance ou de ré-ingénierie. La présentation d'une catégorie de variables de configuration comporte donc un nom, un critère de reconnaissance et plusieurs observations et pistes pour traiter ce type de variables dans le futur. Il est également important de noter qu'une variable ne peut appartenir qu'à une ou deux catégories et, dans le cas de deux catégories, l'une des deux doit être la catégorie *Exclusion*.

La classification a été construite à partir de différents éléments dans les distributions des différentes métriques. Tout d'abord, si l'on considère la distribution du nombre de composants par variable (N_{CV}) et du nombre de lignes par composant et par variable(N_{LCV}), on trouve dans la distribution des variables concentrées sur un composant, d'autres éparpillées sur plusieurs ou encore certaines focalisées sur un composant mais possédant un peu de contrôle sur d'autres. De la même manière, dans la distribution du nombre de lignes par variables (N_{LV}), on peut trouver des variables contrôlant très peu de lignes, d'autre un nombre plus

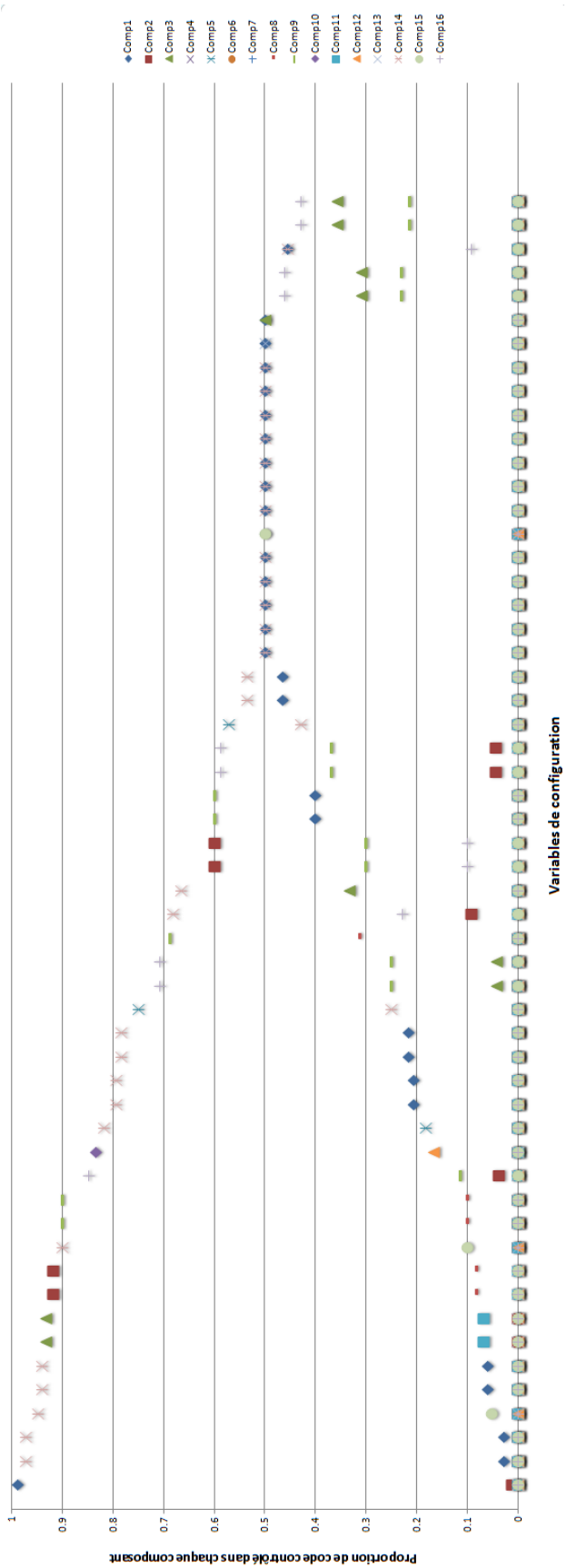

Figure 3.12 Proportion de lignes de code dans chaque composant pour les variables contrôlant 3 composants.

conséquent et certaines tellement qu'elles ne sont plus comparables aux autres. C'est en combinant ces types de comportements vis-à-vis des lignes et des composants que nous avons construit la classification.

- ***Global*** : Les variables de type *Global* sont des variables de haut niveau avec un fort impact sur le code. À partir des distributions de métriques, cela correspondrait à un nombre de lignes de code beaucoup plus grand que la moyenne et un éparpillement sur un bon nombre de composants. Le critère que nous avons établi est donc un contrôle dans plus de la moitié des composants et un nombre de lignes de code contrôlées supérieur à la moyenne de la distribution par plus de deux écarts types. Les variables de cette catégorie présentent plusieurs intérêts et inconvénients pour la gestion de variabilité du FMS. Tout d'abord, elles permettent d'avoir une vision à haut niveau du système et de mieux comprendre le découpage du logiciel. Cependant, lorsque le contrôle qu'elles exercent devient trop grand, le problème se pose de savoir si l'objectif est de contrôler un aspect du logiciel ou de rassembler des fonctionnalités sans raison sous-jacente.
- ***Tiny*** : Les variables *Tiny* sont les variables contrôlant 25 lignes de code ou moins dans un unique composant, ce sont donc des variables concentrées avec peu de lignes de code. Le critère de 25 lignes a été défini par rapport à la médiane de la distribution de N_{LV} qui est 26. La FCA permet de les identifier plus facilement puisqu'il suffit d'étudier les variables réparties sur un unique composant, donc au niveau 1 du treillis Variables-composants. Les variables de cette catégorie contrôlent des aspects extrêmement spécifiques du système, par conséquent, les opérations de maintenance sur ce type de variables devraient nécessiter moins de travail que d'autres puisqu'il y a moins de dépendances. De plus, pour une ré-ingénierie, le code contrôlé par ces variables devrait être plus facilement extrait et intégré dans le module choisi.
- ***Focused*** : Les variables *Focused* correspondent à des variables contrôlant plus d'un composant mais dont les lignes contrôlées sont inégalement réparties, un composant rassemblant la majorité. Il s'agit donc des variables pour lesquelles plus de 50% du code contrôlé est dans un seul composant mais ayant également une portion du code dans d'autres composants. Les variables *Focused* semblent contrôler une fonctionnalité donnée au sein de leur composant principal, tout en contrôlant des détails d'exécution dans d'autres composants pour conserver une cohérence dans le système. Toutefois, cette organisation implique que ces fonctionnalités sont disséminées au sein du FMS. Afin de simplifier les opérations de maintenance et de ré-ingénierie, il serait intéressant de rassembler le code dépendant d'une variable *Focused* dans un même module afin que tout changement sur la fonction principale soit plus simple à répercuter sur le code dépendant.

- ***Aspect*** : Si les variables *Focused* sont centrées sur un composant, les variables *Aspect*, au contraire, sont plus équitablement réparties. Ce sont donc les variables contrôlant plusieurs composants mais sans qu'aucun de ces composants ne rassemble plus de 50% du code contrôlé. Les variables *Aspect* étant réparties sur plusieurs composants, elles se retrouveront dans les niveaux supérieurs du treillis FCA variables-composants. Contrairement aux variables *Focused*, les variables *Aspect* ne contrôlent pas une fonction particulière mais maintiennent la cohérence du système en assurant que de éléments de code incompatibles ne s'exécutent pas en même temps.
- **Exclusion** : Les variables Exclusions sont des variables dont le contrôle sur le code est beaucoup plus restrictif que permissif, c'est à dire que plus de 70% du code contrôlé l'est avec le motif de contrôle LOSS. Le seuil de 70% a été déterminé en considérant que c'est autour de 70% que le seuil est le plus stable. Il est important d'identifier les variables dont la principale utilisation est d'empêcher l'exécution de code car lors d'une ré-ingénierie en ligne de produits, ces variables devraient être éliminées du système, du moins pour le code dont elles interdisent l'exécution. En effet, si le nouveau système se fonde sur des modules que l'on insère seulement s'ils sont pertinents, l'utilisation de la configuration pour interdire l'exécution de code devient caduque.
- ***Parameter*** : Les variables de configuration de type *Parameter* n'exercent aucun contrôle sur le code, mais sont utilisées pour réaliser certains calculs. On peut donc les identifier comme les variables présentes dans le code du logiciel mais apparaissant au niveau zéro du treillis FCA variables-composants.
- **Obsolete** : Comme le type *Parameter*, les variables de type *Obsolete* n'ont aucun contrôle sur l'exécution du code, mais en plus, n'apparaissent à aucun moment dans le système. Les variables *Obsolete* devraient être supprimées du système puisqu'elle ne participent pas à son fonctionnement et compliquent inutilement le système. Les variables *Obsolete* se trouvent donc dans le concept minimum du treillis variables de configuration-produits

Plusieurs observations sont faisables en étudiant la distribution des variables dans les diverses catégories, présentée dans le tableau 3.3. On peut tout d'abord remarquer que la grande majorité des variables est utilisée comme paramètres plutôt que pour contrôler du code. Une ré-ingénierie du système devrait donc prévoir d'étudier la possibilité de conserver un fichier de configuration pour garder une trace de ces paramètres une fois le système réorganisé en modules.

Dans un second temps il est nécessaire de s'intéresser à l'unique variable globale du système, en effet, cette variable contrôle à elle seule près de 30 000 lignes de code dans le système (voir figure 3.11, une seule variable contrôle plus de 10 000 lignes de code), ce qui pose

Tableau 3.3 Nombre de variables dans chaque catégorie

Categories	**Nombre de variables**
Global	1
Tiny	57
Aspect	30
Focused	119
Exclusion	41
Parameter	473
Obsolete	3

certains problèmes en terme de conception. Il est désormais difficile de savoir si la variable contrôle une fonction ou représente au contraire un élément qui devrait être un composant à part entière mais a été divisé au sein du FMS. L'étude du code contrôlé par cette variable par des experts devrait permettre soit de réorganiser le code en un nouveau composant, soit de diviser la variable en plusieurs nouvelles avec des fonctions bien identifiées.

On peut également considérer le grand nombre de variables *Focused* au sein du système. Afin de simplifier les traitements il pourrait être intéressant de distribuer le code contrôlé par ces variables en une variable avec une fonction précise et une autre chargée d'harmoniser le fonctionnement du système en tant que variable *Aspect*. Il serait également envisageable de tenter de tout rassembler en un unique point à considérer comme un module de la future ligne de produits.

Le nombre de variables *Aspect* est assez restreint. En fonction de leur rôle dans le système, qui devra être confirmé par des experts du logiciel, il serait intéressant de tenter de regrouper les fonctionnalités qu'elles représentent chacune de leur coté pour créer un module pouvant se greffer à tout produit, ou, au contraire, d'inclure chacun des aspects de chaque variable dans le module le plus pertinent pour chacun des produits concernés.

Il est selon nous important de remarquer le nombre non négligeable de variables *Exclusion*. En effet, ce type de variable ne devraient pas être conservé dans un système organisé en ligne de produits, pour la simple raison que ce type de système ne devrait contenir que du code pertinent à son fonctionnement propre et non, comme c'est le cas pour un système à configuration dynamique, l'ensemble du code du système, qu'il est donc nécessaire de filtrer.

Après avoir conçu cette classification tenant compte des observations du système, nous avons présenté la répartition des variables au sein des catégories à un expert du système. Ses observations nous ont permis de confirmer plusieurs avantages de la classification mais aussi de définir certains axes d'amélioration pour des travaux futurs.

Dans un premier temps, l'expert nous a confirmé que la catégorie *Global* représente en

effet une variable dont l'influence sur le système est énorme, mais qui, à l'heure actuelle, ne peut pas encore être subdivisée pour faciliter la maintenance et l'évolution.

La classification présente des comportements typiques des variables de configuration au sein du système. Elle permet également, si l'on connait le comportement attendu d'une variable, d'identifier des comportements qui ne sont pas optimisés par rapport au rôle théorique d'une variable au sein du système. Dans cette optique, nous avons pu avoir confirmation, par l'expert avec qui nous sommes en contact, que les catégories *Focused* et *Aspect* permettent d'identifier des variables dont la distribution sur le code n'est pas optimisée par rapport à la fonction qu'elles représentent. Certaines devraient être des aspects et se retrouvent concentrées sur un composant, tandis que d'autres incarnent la situation inverse. Bien que cela ne nuise pas au fonctionnement du logiciel (les normes de tests assurent un comportement correct du système), il s'agit néanmoins de situations pouvant être optimisées afin de faciliter une future ré-ingénierie.

L'expert nous a également confirmé que les variables entrant dans la catégorie *Parameter* sont des variables utilisées pour de simples calculs ponctuels plutôt que pour du contrôle de code. La vérification de cet état de fait nous a toutefois fourni la confirmation que cette catégorie est intéressante du point de vue de l'entreprise, aucune liste de ces variables n'étant actuellement disponible.

Les axes d'amélioration proposés par l'expert ayant étudié l'appartenance des variables aux catégories se situent à plusieurs niveaux. Tout d'abord, certains raffinements sont possibles au sein des variables *Aspect*, par exemple entre les variables séparées sur deux composants à parts égales et celles éparpillées plus nettement avec de forte variations dans la proportion de code dans chaque composant ou encore pour séparer les variables contrôlant beaucoup de code de celles contrôlant seulement quelques lignes dispersées. D'autres différenciations seraient possibles sur les variables *Focused* ou même *Tiny*. L'idée serait d'identifier des sous-catégories eu sein de la typologie définie ici de manière à prendre en compte d'autre critères comme la dispersion du contrôle sur les fichiers en plus des composants.

En conclusion, les catégories de variables de configuration permettent d'une part de mieux comprendre le système mais aussi de pointer des éléments sur lesquels porter une attention plus soutenue en cas de maintenance ou de ré-ingénierie. C'est également un moyen d'identifier certaines mauvaises habitudes d'implantation, identifier par exemple qu'une fonction aurait dû être concentrée dans un composant au lieu d'être dispersée ou au contraire, que le contrôle de plusieurs éléments aurait dû être réparti. Certaines améliorations restent à considérer, notamment dans l'étude des partages de contrôle, et nous nous y intéressons en partie dans le chapitre 5.

3.4.2 Interactions entre variables

De la même manière que nous avons caractérisé les types de variables de configuration, nous avons souhaité identifier des relations typiques entre variables. Les types de relations permettent de connaitre l'impact qu'une variable peut avoir sur les autres lorsque l'on modifie du code pour y correspondre ou qu'on fait évoluer la valeur de la variable. Comme précédemment, la présentation d'une relation entre variables se compose d'un nom, d'un critère d'identification formel et d'observations quand à ce qui peut en être déduit pour de futures évolutions.

- ***Associated*** : Deux variables sont considérées comme ayant une relation *Associated* lorsqu'elles contrôlent une ligne de code commune avec le même motif. Formellement, avec P l'ensemble des motifs et v_1, v_2 deux variables. v_1 et v_2 sont *Associated* si et seulement si :

$$\exists p \in P | p(v_1) \cap p(v_2) \neq \emptyset \tag{3.8}$$

  Une grande partie des couples de variables *Associated* sont identifiable facilement au sein du treillis FCA entre variables de configuration et code contrôlé (cf figure 3.9), ce sont les variables appartenant à au moins un concept du treillis en commun. La recherche sur les autres motifs (LOSS, SOMEHOWPLUS et SOMEHOWMINUS) se fait plus simplement car le nombre de lignes contrôlées est nettement plus faible. De plus, deux variables ne peuvent avoir cette relation que si elles appartiennent à un concept commun dans le treillis de FCA entre variables et composants, deux variables ne partageant pas de composant ne pouvant pas partager de code. Plus de détails pourront être trouvés au chapitre 5.

  L'identification des variables contrôlant du code en commun avec le même motif permet d'identifier d'éventuels recoupements de contrôle entre variables, mais aussi d'identifier une forme de hiérarchie de contrôle avec des inclusions des contrôles au sein de blocs de code dont la taille est réduite.

  De plus, deux variables contrôlant du code en commun sont supposées agir sur des fonctionnalités similaires. Une vérification de cette propriété sur les couples de variables peut aider à identifier de mauvais choix de conception et à les réparer.

- ***Complementary*** : Deux variables ont une relation *Complementary* lorsqu'elles contrôlent la même ligne de code avec deux motifs de contrôle différents. Soit, avec les notations précédentes, deux variables sont *Complementary* si et seulement si :

$$\exists p_1, p_2 \in P | p_1 \neq p_2, p_1(v_1) \cap p_2(v_2) \neq \emptyset \tag{3.9}$$

Les couples de variables reliés par la relation *Complementary* permettent d'identifier des relations d'opposition entre variables de configuration. Par exemple une variable peut amener l'exécution d'une ligne de code avec le motif GAIN tandis qu'une autre peut l'empêcher avec le motif LOSS. Il est important de reconnaitre ces relations car des variables s'opposant sur l'exécution du code sont a priori responsables d'harmoniser le comportement des éléments du FMS entre eux. Par conséquent, de telles variables devraient appartenir à des modules différents lors d'une ré-ingénierie.

– ***Dependant*** : Enfin, une variable v_1 est considérée comme dépendante d'une autre variable v_2 lorsqu'il n'existe aucun produit possédant v_1 sans v_2. Cette relation est extraite directement à partir des implications de Duquenne-Guigue au sein du treillis variables-produits, c'est-à-dire qu'une variable est impliquée par une autre lorsqu'il n'existe pas de concept du treillis tels que la variable dépendante y est présente sans la variable impliquée. La relation de dépendance étant fondée sur la relation d'implication. La relation de dépendance présentée ici est définie de manière extrêmement formelle et ne peut donc être utilisée telle quelle. L'intérêt de cette relation est d'identifier une forme de hiérarchie au sein des variables, certaines combinaisons étant impossibles ou au contraire forcées pour des raisons intrinsèques au système mais non documentées. Cette relation est donc un premier filtre pour identifier de telles dépendances, un tri devant toutefois être fait par un expert.

Une première remarque que l'on peut faire au sujet de ces types de relation est que les relations *Associated* et *Complementary* peuvent être vues comme des relaxations de deux des relations identifiées par Loesch *et al.* [43] que sont "only used in pairs" et "only used mutually exclusively". Cependant, la confrontation à un système industriel de pointe a mis en évidence le fait que les définitions très restrictives des relation "Only used in pair" et "Only used mutually exclusively" ne possèdent pas d'instances dans le système et ne permettent donc de mettre en évidence aucun comportement existant dans notre cas d'étude.

Le tableau 3.4 récapitule le nombre d'instances de chacune des relations que l'on trouve au sein du FMS. Tout d'abord, il est important de noter que les relations entre variables ne sont pas exclusives dans leur définition, il est donc normal que le nombre de relations

Tableau 3.4 Nombre d'instances de relations

Relations	**Nombre d'instances**
Associated	455
Complementary	377
Dependant	808

soit très grand. Nous avons trouvé plusieurs instances de couples de variables étant à la fois *Associated* et *Complementary*, ce qui peut identifier deux types d'interactions : dans un premier cas, il peut s'agir d'une imbrication du contrôle des variables, c'est à dire que l'une des variables n'est testée que si l'autre est dans un état donné, et ce cas de figure est voulu dans le contexte avionique car fortement vérifiable et traçable. C'est l'exemple fourni en figure 3.13, le code dans la branche *else* de $v2$ est contrôlé en GAIN par $v1$ et en LOSS par $v2$. A l'inverse, il est possible que les deux variables partagent ces deux relations sans cette relation d'imbrication, et il est alors nécessaire d'évaluer si cette interaction est volontaire ou s'il s'agit d'une implémentation non optimale d'une fonction.

Par ailleurs, le grand nombre de relations *Dependant* au sein du système prouve qu'un tri par un expert, pour identifier les dépendances réelles et les relation purement contingente, est nécessaire. L'identification de ces relations permet toutefois de n'éliminer aucune dépendance potentielle, et présente l'intérêt de ne pas éliminer les dépendances non documentées.

Lors de la présentation des types d'interactions à l'expert du système, ses commentaires nous ont permis de tirer plusieurs conclusions. Tout d'abord, l'expert a confirmé que les oppositions peuvent provenir de différentes structures de code. Il nous a donc suggéré de tenter d'identifier les différents cas possibles afin de définir des stratégies appropriées pour résoudre les situations non optimisées. C'est à cette problématique que s'intéresse le chapitre 5.

En conclusion, dans ce chapitre, nous avons introduit les données disponibles à l'origine du projet ainsi que les différentes analyses effectuées. Nous avons donc présenté les différents treillis FCA obtenus au cours du projet ainsi que de nouvelles métriques permettant d'avoir une meilleure compréhension du système. Enfin, nous avons introduit une classification originale des variables de configuration et des relations induites par leur utilisation dans le système. Cette classification s'accompagne de plusieurs remarques portant sur des

```
if(v1){
   ...
   if(v2){
      ...
   }else{
      ...
   }
   ...
}
```

Figure 3.13 Exemple d'inclusion de contrôle

pistes à étudier pour une ré-ingénierie ou une maintenance du FMS. Nous avons également obtenu une validation de nos résultats par un expert du système, les catégories de variables permettent non seulement de connaitre la composition du FMS mais aussi d'identifier les utilisations non optimisées de variables par rapport à leur rôle dans le logiciel. Enfin, nous avons pu définir plusieurs éléments d'amélioration de l'approche à envisager pour des travaux à venir, tant dans la définition de notre classification que dans les techniques à envisager pour la maintenance et la ré-ingénierie.

CHAPITRE 4

Base de données de résultats

Après avoir effectué l'ensemble des analyses présentées dans le chapitre précédent, la problématique a été posée de savoir comment conserver ses résultats tout en les rendant interactifs. Dans ce chapitre nous présentons l'ensemble des solutions envisagées, la solution implémentée ainsi que différentes tâches ayant pu être accomplies grâce à la base de données. Nous nous intéressons également aux futurs travaux envisageables à partir des données intégrées à la base de connaissances.

4.1 Contraintes

La création d'une base de connaissances pour représenter les résultats des analyses menées était contrainte par différents facteurs qui ont orienté le choix de structure final. Dans cette section nous présentons ces contraintes et les circonstances qui les amènent.

La première contrainte imposée à la base de données est bien entendu la possibilité d'interagir et de visualiser facilement les résultats disponibles. Cela s'explique d'une part par le besoin de consultation à titre de documentation, mais aussi parce que les résultats mettent en évidence certains comportements et caractéristiques pouvant servir lors d'opérations de maintenance et de ré-ingénierie. Il est donc nécessaire d'avoir une base de données interactive, mais il est également important de choisir une implémentation pour laquelle les interactions sont simples à prendre en main pour un utilisateur sans expertise au niveau des résultats et du système.

La seconde contrainte concerne l'extensibilité de la base de données. En effet, l'idée directrice dans cette base de connaissances est de rassembler les connaissances relatives au système et pas seulement les résultats des présents travaux. Il était donc nécessaire de trouver une implémentation pouvant facilement supporter des extensions de données ou de modèles.

Une autre contrainte à laquelle nous avons été confrontée concerne la possibilité de bâtir des inférences au sein du système. En effet, pour une opération de ré-ingénierie, les notions de relation entre éléments et de traitement systématique de ces relations sont importantes. Il était donc important de pouvoir ajouter ces éléments à partir de règles simples.

Enfin, la solution choisie devait se fonder sur un standard reconnu afin de disposer non seulement d'une documentation fournie mais aussi de nombreux outils permettant de changer de support d'implantation sans remettre en cause la base de données elle-même.

En résumé, les contraintes imposées pour la base de connaissances se résument à la facilité de récupération d'information, l'extensibilité de la structure, la possibilité de bâtir des inférences et la forme de standard reconnu.

4.2 Données à intégrer

Dans cette section, nous présentons les données du projet qui devaient être incluses dans la base de connaissances. En effet, toutes les données extraites n'ont pas besoin d'être intégrées dans la mesure où une partie des résultats doit pouvoir être retrouvée ou recalculée en cas d'évolution importante du système.

4.2.1 Données initiales

Les données initiales du projet devaient presque toutes être intégrées dans la base de connaissances dans la mesure où il fallait pouvoir mener de nouvelles analyses et pouvoir les lier si de nouvelles relations étaient identifiées. Notons que les données relatives au graphe de flux de contrôlei, ayant déjà été exploitées dans leur totalité, ne sont pas véritablement nécessaires.

Il fallait donc inclure, d'une part la liste des composants du système avec, pour chacun, la liste des fichiers de code rattachés, ce qui implique une certaine forme de graphe, et d'autre part les variables de configuration à la base des présents travaux. Bien entendu, il était également important d'intégrer les lignes de code contrôlées pour chacune des variables en mentionnant le motif de contrôle.

Sur l'ensemble de ces données, il est important de remarquer qu'il s'agit plus de conserver l'information relative à des relations entre éléments qu'à des valeurs de métriques proprement dites. Cette idée a joué un rôle important dans le choix de la solution implémentée.

4.2.2 Résultats

Au delà des données initiales du projet, il était également important de conserver les données résultant des analyses afin de pouvoir les réutiliser dans d'autres analyses. Dans ce contexte, il était nécessaire de conserver l'ensemble des treillis résultant des différentes analyses de FCA, et ce afin d'avoir accès aux différentes hiérarchies mises en évidence par la FCA.

La conservation des métriques posait un problème différent : stocker le résultat directement permet de conserver la valeur, mais les données déjà incluses fournissaient tout le nécessaire pour pouvoir les calculer. Il semblait donc plus pertinent de ne pas inclure directement ces résultats afin de limiter l'occupation mémoire et de chercher une implantation

supportant le calcul des différentes métriques.

Concernant la classification de variables présentée au chapitre précédent, son stockage semblait pertinent et nous avons donc cherché une implantation supportant le stockage d'une part, de la typologie et, d'autre part, des relations entre variables.

Comme précédemment, on peut remarquer que les éléments à conserver relèvent plus de structures relationnelles ou de structures de graphes que de valeurs fixées. C'est donc un type d'information particulier qui influence le choix d'implantation fait.

4.3 Solutions envisagées

La présente section introduit les deux principales pistes envisagées pour implanter la base de connaissances avec les avantages et inconvénients de chacune par rapport aux données et aux contraintes imposées. Quoique la description des possibilités soit ici sommaire, de plus amples informations relatives à la solution choisie sont fournies dans les sections suivantes. Nous expliquons également le choix qui a été fait au regard des contraintes et intérêt de chacune des alternatives.

4.3.1 BDD relationnelle

La première solution ayant été envisagée était l'utilisation d'une base de données relationnelle, utilisée très largement dans tous les domaines de l'ingénierie, et présentant de nombreux avantages.

Si l'on s'intéresse au respect des contraintes fixées, on peut tout d'abord considérer la simplicité d'interrogation d'une telle base de données relationnelle. En effet, le langage d'interrogation SQL et ses dérivés sont largement connus et utilisés par la plupart des développeurs et il ne représente donc aucune difficulté de prise en main.
Par ailleurs, les bases de données relationnelles peuvent être facilement étendues par l'adjonction de nouvelles tables. Toutefois, pour garantir l'extensibilité facile de notre système il serait nécessaire de prêter une grande attention à la conception du schéma de base de données.
Enfin, l'une des principales difficultés posées par les bases de données relationnelles concerne la construction d'inférences. En effet, ce type de requête prend une forme complexe et pose des problèmes dûs à la conception des tables dans la base de données, le respects des liens existant pouvant être brisé par de nouvelles inférences.

On peut ensuite s'intéresser à la pertinence des bases de données relationnelles pour conserver les données voulues. Il est important de noter que les connaissances à conserver sont de types relationnels, et que dans une base de données relationnelle, ces informations

sont stockées chacune dans une table différente et que l'extension de la base de connaissances impliquerait nécessairement des modifications dans le schéma de la base de données.

4.3.2 Base de données de graphes RDF

La seconde solution considérée a été l'utilisation d'une technologie du web sémantique basée sur le standard RDF. Cette idée a été inspirée par la nature relationnelle des données à conserver et la structure de graphe d'une grande partie de ces données. Le standard RDF de bases de données de graphes et le langage d'interrogation SPARQL sont des normes publiées à l'origine en 1999 par le W3C [52] afin de représenter le web sémantique. De nombreuses organisations se servent de ce format pour présenter des données, comme par exemple le gouvernement anglais, Wikipedia ou Mozilla. Dans la mesure où il s'agit d'une structure moins répandue que les bases de données relationnelles, le paragraphe suivant présente les principales caractéristiques de ce paradigme et des langages qui l'utilisent, comme c'est le cas de SPARQL. Nous nous intéressons également au positionnement de cette solution dans le contexte qui est le nôtre.

Définition de RDF :

La structure de base de données de RDF est prévue pour organiser des informations structurées en graphe, c'est à dire des relations entre objets, que la relation induite soit effectivement une arête entre deux nœuds d'un graphe ou bien un lien quelconque entre les deux objets. C'est ce qui le rend si approprié pour la représentation de relations entre éléments plutôt que le stockage de valeurs. Le standard RDF comprend alors trois types d'éléments qui sont :

- Les ressources : tous les éléments pouvant appartenir à la base de données. Dans notre contexte, cela inclut les concepts des treillis FCA, les lignes de code ou encore les variables de configuration.
- Les propriétés : les types de données, propriétés ou encore les relations possibles au sein de la base de données. Dans notre cas d'étude, cela inclut les arêtes des treillis FCA, les liens de contrôle entre variables et ligne de code ou bien l'appartenance d'un fichier à un composant.
- Les triplets : ce sont des assertions créant une relation entre deux ressources à l'aide d'une propriété. Pour $Sujet$ et $Objet$, deux ressources de la base de données et pour $Predicat$, une propriété, le triplet de relation intégré à la base de données est présenté en 4.1.

$$Sujet, Predicat, Objet \tag{4.1}$$

Il est également possible d'introduire des données brutes telles que des chaînes de caractères ou des nombres dans une base RDF sous la forme de littéraux, cependant, de tels littéraux ne seront trouvés que par les relations qui les lient et ne peuvent être utilisés comme sujets dans un triplet. Il sera plus complexe d'effectuer des requêtes les concernant. Il est important de noter que le standard RDF impose à tout éléments, ressource ou propriété, de prendre la forme d'une URI et ce, quelle que soit la sérialisation choisie. Une sérialisation étant une représentation textuelle de l'ensemble des relations d'une base de données de graphe.

Le standard RDF impose certains types de contraintes sur les classes d'éléments autorisées et sur les propriétés permises, toutefois la nature de ressources et la création de nouveaux types est laissée à l'entière liberté des utilisateurs. L'ensemble de la dernière version du standard est définie dans [53].

Par ailleurs, en raison de la notation d'URI pouvant être lourde et à la fois peu lisible et difficilement manipulable, il est possible de définir des préfixes de notation facilitant l'usage de RDF. Ainsi, si un usage récurrent est fait des URI débutant avec :
"http ://www.w3.org/1999/02/22-rdf-syntax-ns#", il est possible d'associer ce début d'URI avec le préfixe "rdf" et la notation pour un élément *elem* sera alors : rdf :$elem$

De plus, le standard RDF supporte plusieurs formats de sérialisation permettant de s'adapter aux différents besoins des utilisateurs. Le premier et plus ancien est une sérialisation au format RDF/XML. Cependant, si RDF/XML apporte un formalisme important pour la sérialisation, les contraintes qu'il impose peuvent poser des problèmes de lisibilité pour un utilisateur humain. D'autres formats fondés sur la représentation de triplets purs et simples existent et sont aujourd'hui préférés par la plupart des utilisateurs. On peut ainsi dénombrer les formats Turtle, N-Triple, JSON-LD et d'autres, tous acceptés par le standard du W3C.

La description d'une application du web sémantique est souvent associée avec la définition d'une ontologie permettant de définir le type de graphe que l'on peut obtenir. En effet, en lui même, le standard RDF permet tous les triplets possibles avec les ressources et propriétés introduites dans la base de données. Toutefois, si l'on souhaite vérifier que les données ont été effectivement rentrées, que les liens entre éléments sont valides, il peut être bon de définir formellement les types de relation que l'on souhaite autoriser ou même imposer. C'est dans ce contexte que l'on peut définir une ontologie, qui sert de modèle pour visualiser les relations possibles entre chaque couples de ressources. On peut souvent présenter une ontologie comme un graphe recensant les relations possibles dans un ensemble de données et il est en général plus pertinent d'aboutir à un graphe connexe, chaque nœud représentant une ressource d'un type particulier et chaque arête représentant une propriété.

Possibilités de SPARQL :

En association avec le modèle de données RDF, le W3C a standardisé le langage d'interrogation des bases de données RDF avec le langage SPARQL (*SPARQL Protocol and RDF Query Language*). Cette normalisation est apparue en janvier 2008 avec une nouvelle version en 2013. Le langage SPARQL s'appuie largement sur des structures classiques de bases de données avec des mots clés tels que SELECT, WHERE, LIMIT, ORDER BY, etc... Mais de nombreuses différences apparaissent. Par exemple, la recherche d'un élément ne se fait pas par sa colonne dans une table donnée mais par les différentes relations qu'il partage avec d'autres. L'intérêt du langage est que la nature intuitive de la représentation en triplets facilite l'expression des requêtes pour un utilisateur.

Le langage SPARQL s'articule autour de quatre types de requêtes visant des objectifs différents et qui sont les suivants :

- **SELECT** : Comme en langage SQL, les requêtes de type SELECT visent à la pure récupération d'information. Il faut déterminer quelles ressources on souhaite extraire et la base de données retourne un tableau contenant les instances validant les conditions de la requête.
- **CONSTRUCT** : C'est une requête visant à construire un graphe RDF à partir d'un motif de graphe à identifier. La sortie sera au format RDF standard et les solutions ne respectant pas le standard RDF sont éliminées.
- **ASK** : Les requêtes de type ASK sont de la même forme que les requêtes SELECT, mais le retour se fait sous la forme d'un booléen de valeur TRUE s'il y a au moins une solution et FALSE sinon.
- **DESCRIBE** : Les requêtes de type DESCRIBE fournissent un graphe décrivant les données vérifiant le motif conditionnel fourni. Le choix de la description dépend de l'implémentation de la base de données, le standard n'étant pas clair à ce niveau.

On peut trouver de nombreuses implantations de bases de données RDF prenant le langage SPARQL comme langage de requête. La page Wikipedia du langage rassemble à elle seule plusieurs dizaines de systèmes implémentant le langage SPARQL parmi lesquels on peut trouver Allegro[54], 4store [55] ou encore Sesame [56]. Certains de ces systèmes viennent avec une structure de logiciel, d'autre comme une implantation en tant que serveur avec une interface web ou même une API pour un ou plusieurs langages.

Positionnement en tant que solution :

Si l'on compare cette solution aux contraintes qui s'appliquent à notre projet, on peut

tout d'abord considérer que le format RDF et le langage SPARQL sont nouveaux et pourraient poser des difficultés d'apprentissages aux utilisateurs. Cependant, les similarités des requêtes entre les langages SQL et SPARQL permettent d'assurer une facilité d'apprentissage et d'utilisation proche de celle d'une base de données relationnelle.
Par ailleurs, le format de triplets de RDF permet une extensibilité très grande, la seule contrainte imposée étant de devoir notifier des relations entre éléments et éventuellement des restrictions sur les relations autorisées.
Enfin, la structure de RDF et du langage SPARQL, utilisant entre autre des requêtes de type CONSTRUCT, permet d'insérer de nouvelles relations entre éléments simplement en formalisant les motifs de graphe auxquels un élément doit appartenir pour subir l'ajout de relations, ce qui permet donc facilement l'application d'inférences.

Le problème se pose alors de choisir entre les deux solutions possibles. S'oppose alors la facilité d'utilisation à la pertinence de la solution par rapport aux données et à l'application future d'inférences. Dans la mesure où l'objectif des présents travaux visent à aider de futures évolutions du système et des choix de ré-ingénierie, nous avons décidé de privilégier le choix d'une base de donnée RDF avec une interface d'interrogation SPARQL. Ce choix s'appuie également sur l'idée que la base de données n'est qu'un premier pas dans la documentation de la variabilité du système et que l'évolution de la structure d'une base de données RDF est nettement plus simple à mettre en œuvre que la re-conception d'une base de données relationnelle. Les choix d'implantation qui ont été faits sont détaillés dans la section suivante.

4.4 Implantation de la base de connaissances

Dans cette section, nous présentons la conception et l'implantation de la base de données dans le contexte de l'entreprise partenaire. Cela inclut le choix de plateforme de la base de données, la définition de l'ontologie et d'une interface d'interrogation. Nous présentons également quelques requêtes SPARQL ayant permis de trouver des réponses à des interrogations de notre partenaire industriel.

4.4.1 Sesame

Le choix du système d'implantation s'est fait par rapport à certaines contraintes imposées au projet en termes de coût mais aussi de besoins. L'objectif étant de rendre le système disponible pour des analystes travaillant éventuellement sur des sites différents sans nécessiter une adaptation totale du système, le besoin a été posé d'une interface web avec accès au système via internet, des normes de sécurité pouvant imposer un accès par VPN au besoin, la taille des données à intégrer et d'éventuels futurs besoins imposent également la disponibilité

d'une API de programmation dans un langage courant tel que C ou Java. Enfin, le choix d'un système distribué gratuitement est intéressant en terme de coût, avec la nécessité d'une licence autorisant une utilisation commerciale. Après avoir étudié les caractéristiques de plusieurs systèmes, notre choix s'est porté sur le système Sesame 2, distribué par OpenRDF.org répondant à tous les besoins y compris une API de programmation en Java.

La seule limite de Sesame se situe dans les temps de calcul, plus long que certains systèmes commercialisés, mais le type d'utilisation, non critique et non urgente, rend la gratuité plus intéressante que le temps de réponse du système.

Sesame est conçu comme un serveur de base de données auquel on peut accéder soit par interface web, soit par API de programmation. L'une des distributions, adaptée aux machines sous Linux est prévue pour un déploiement sur un serveur apache, c'est la solution que nous avons choisie car c'est la plateforme de fonctionnement prévue dans l'entreprise.

4.4.2 Définition des éléments

Nous définissons ici les ressources, propriétés et triplets utilisés dans la base de données mise au point. Il est important de noter que l'ontologie qui découle de ces définitions n'est aucunement définitive. En effet, de futurs besoins d'identification de relations ou de mesures pourront faire apparaître de nouvelles ressources et propriétés. La figure 4.1 présente l'ontologie à la base du système en incluant uniquement les données présentées dans le chapitre 3. Nous utilisons le préfixe "fca" pour représenter la base des URI utilisées. Dans notre définition, ID représente l'identifiant d'une ressource en particulier tandis que la figure 4.1 répertorie les interactions entre classes. On peut alors définir notre web sémantique de la manière suivante :

1. Ressources : Les ressources correspondent aux différents éléments analysés ou résultants des analyses effectuées et sont les suivantes :
   - Ligne de code - `fca:Line\#ID`
   - Fichier - `fca:File\#ID`
   - Numéro de ligne - `fca:LineNumber\#ID`
   - Sous-Système - `fca:SubSystem\#ID`
   - Composant-`fca:Component\#ID`
   - Produit - `fca:Product\#ID`
   - Variable de configuration - `fca:ConfVar\#ID`
   - Nœud de FCA - `fca:Node\#ID`
   - Catégorie de typologie - `fca:Category\#ID`
2. Propriétés : Les propriétés correspondent à des relations qui seront mieux comprises à partir des triplets. Toutefois, il est important de noter que les termes "MOTIF" et

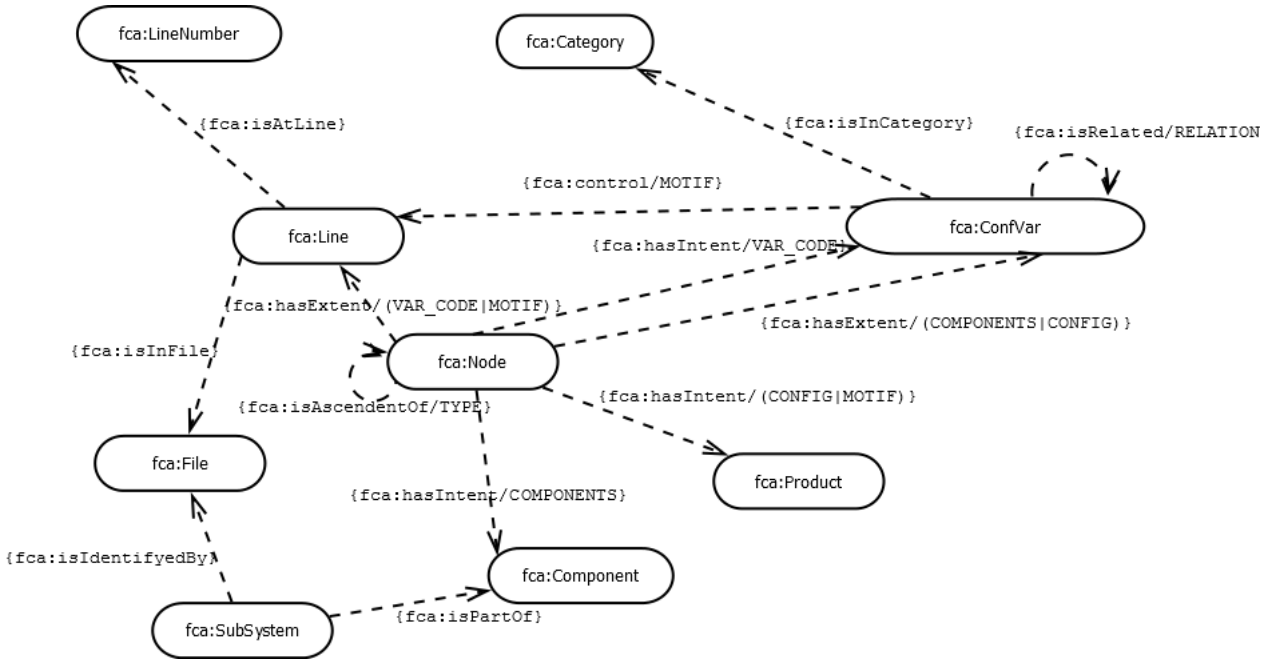

Figure 4.1 Schéma de l'ontologie implantée

"TYPE" représentent un élément quelconque respectivement de {GAIN, LOSS, SOMEHOWPLUS, SOMEHOWMINUS} et {VAR_CONFIG, GAIN, LOSS, SOMEHOWPLUS, SOMEHOWMINUS, VAR_CODE, VAR_COMPONENT}. "MOTIF" énumère les motifs de contrôle tandis que "TYPE" liste les différentes analyses FCA effectuées. De même, "RELATION" représente l'une des relations de {ASSOCIATED, COMPLEMENTARY, DEPENDENT}

- `fca:isAtLine`
- `fca:isInFile`
- `fca:isIdentifiedBy`
- `fca:isPartOf`
- `fca:control/MOTIF`
- `fca:inCategory`
- `fca:isRelated/RELATION`
- `fca:hasIntent/TYPE`
- `fca:hasExtent/TYPE`
- `fca:ascendentOf/TYPE`

3. Triplets : A niveau des triplets, il est important de noter que "intent(TYPE)" et "extent(TYPE)" représentent respectivement les classes servant d'intension et d'extension dans les treillis FCA étiquetée "TYPE" avec la même définition que précédemment.

- `fca:Line\#ID fca:isAtLine fca:LineNumber\#ID`
- `fca:Line\#ID fca:isInFile fca:File\#ID`
- `fca:SubSystem\#ID fca:isIdentifiedBy fca:File\#ID`
- `fca:SubSystem\#ID fca:isPartOf fca:Component\#ID`
- `fca:ConfVar\#ID fca:control/MOTIF fca:Line\#ID`
- `fca:ConfVar\#ID fca:inCategory fca:Category\#ID`
- `fca:ConfVar\#ID1 fca:isRelated/RELATION fca:ConfVar\#ID2`
- `fca:Node\#ID fca:hasIntent/TYPE intent(TYPE)`
- `fca:Node\#ID fca:hasExtent/TYPE extent(TYPE)`
- `fca:Node\#ID1 ascendantOf/TYPE fca:Node\#ID2`

L'ensemble des définitions précédentes est représenté dans la figure 4.1 où l'on peut observer les types de motifs que l'on peut chercher dans la base de données. L'ontologie présentée a été conçue afin de permettre d'obtenir la plupart des résultats de cette étude à l'aide de requêtes. C'est ainsi le cas des métriques, des caractéristiques permettant d'obtenir les catégories. La base de données implantée représente plus de 500 000 triplets.

4.4.3 Interface Java

Au delà de l'interface web proposée par Sesame, nous avons également souhaité mettre à profit l'API de programmation Java prévue pour interagir avec la base de données par programmation. Nous avons donc conçu une application d'interrogation en Java permettant à la fois l'interaction avec la base de données mais aussi d'autres fonctionnalités détaillées ci-après.

La première fonctionnalité prévue dan l'application est une interface d'interrogation pure et simple, c'est à dire qu'en fournissant une requête SPARQL, on obtient l'ensemble des résultats voulus. Cette interface a pour objectif de fournir un exemple de programmation utilisant l'API pour de futures extensions. C'est également un moyen de pouvoir interroger la base de données en parallèle des autres fonctionnalités sans pour autant avoir à changer d'application. L'API utilisée se présente comme une librairie de fonctions dédiées à l'interrogation d'un serveur Sesame.
La seconde fonctionnalité est un câblage direct d'une requête qui a éveillé l'intérêt des experts de l'entreprise. Il s'agit de connaitre, pour chaque produit, ou éventuellement combinaison de produits, l'ensemble des éléments communs, qu'il s'agisse des variables de configuration ou bien des lignes de code devant être liées au(x) produit(s) selon un motif de contrôle donné.
La dernière fonctionnalité n'est pas reliée à la base de données RDF car, le standard SPARQL n'incluant pas la récursivité des requêtes, l'utilisation d'une programmation en propre s'avérait plus pertinente ; cette dernière fonction est en fait une forme de vérification du contrôle, il

s'agit de savoir, pour une ligne de code, une variable de configuration et un motif de contrôle donnés, à quel point dans le programme se crée la dépendance. En effet, à des fins de refactorisation ou même plus simplement de maintenance, il est nécessaire de connaitre le point de départ d'une dépendance pour pouvoir la modifier ou la supprimer. C'est à ce besoin que répond cette troisième fonctionnalité. Les deux dernières fonctions sont présentées par des formulaires rapides à remplir afin d'éviter de fastidieuses étapes de programmation ou de recherche aux utilisateurs.

Cette interface a été conçue en Java afin de s'appuyer sur la portabilité du langage et permettre son utilisation sur les différentes plateformes utilisées dans l'entreprise.

4.5 Quelques exemples de requêtes

Afin d'illustrer les possibilités de la base de données, nous proposons dans cette section quelques requêtes SPARQL permettant soit de retrouver des résultats, d'obtenir de nouvelles informations ou même de simplement accéder à des informations disponibles pour les utiliser.

4.5.1 Calcul de métrique

Le premier exemple que nous nous proposons d'étudier concerne le calcul d'une des métriques que nous avons mises au point durant nos travaux, il s'agit du nombre de composants par variable. Cette requête repose sur deux types de ressources de la base de données que sont les variables de configuration et les composants du système. Afin de pouvoir définir la requête SPARQL il est nécessaire de trouver le lien entre ces deux ressources dans l'ontologie.

On obtient alors la succession de relations suivante :

- Variable contrôle ligne de code
- Ligne de code appartient à fichier
- Fichier est dans sous-système
- Composant contient sous-système.

L'objectif est donc de comptabiliser, pour chaque variable les composants pouvant être relié car ils contiennent une ligne de code contrôlée. On définit alors la requête SPARQL présentée en figure 4.2.

4.5.2 Interactions entre variables

Une autre requête envisageable peut se porter sur l'interaction entre deux variables $V1$ et $V2$, par exemple pour savoir si elles remplissent les critères pour partager la relation Associated de la typologie présentée dans le chapitre précédent.

```
SELECT DISTINCT ?var (COUNT (DISTINCT ?comp) as ?count)
WHERE{

?var ?control ?line

?line <http://polymtl.ca/FCA/isInFile> ?file
?subsys <http://polymtl.ca/FCA/isIdentifiedBy/> ?file
?subsys <http://polymtl.ca/FCA/isPartOf/> ?comp

FILTER regex(str(?control), ‘‘control/’’ )
}
GROUP BY ?var
```

Figure 4.2 Requête SPARQL pour obtenir le nombre de composants par variable

Cette fois-ci, le chemin reconnaissant le critère est une répétition, $V1$ contrôle la ligne L avec le motif de contrôle M et $V2$ contrôle aussi L avec M. La requête se construit alors en identifiant que $V1$ contrôle L, que $V2$ contrôle L et que les deux contrôles sont identiques. La requête construite est présentée dans la figure 4.3

4.6 Construction d'inférences

Une autre possibilité de la base de données, voulue afin de faciliter à la fois l'utilisation et les évolutions, concerne la construction d'inférences. En effet, avec la structure de RDF et le langage SPARQL, il est aisé de reconnaitre des structures de graphes et d'ajouter de l'information sur des propriétés nouvellement identifiées. Dans cette section, nous présentons une démarche de construction d'inférences que nous avons réalisée puis nous fournissons

```
SELECT DISTINCT ?var1 ?var2
WHERE{

?var1  ?control ?line
?var2  ?control ?line

FILTER regex(str(?control), ‘‘control/’’)
}
```

Figure 4.3 Requête SPARQL pour savoir si deux variables sont Associated

plusieurs pistes pour d'autres inférences à construire.

4.6.1 Relation de spécialisation

Nous avons cherché à identifier une certaine forme de dépendance entre variables dont la définition est proche de la relation Associated présentée au chapitre précédent. Dans le treillis variables-code, nous définissons la relation *spécialisation partielle* de la manière suivante : Pour deux variables $V1$ et $V2$, $V2$ est une spécialisation partielle de $V1$ si et seulement s'il existe deux nœuds $N1$ et $N2$ tels que, avec la notation SPARQL :

- $N1$ `fca:hasIntent` $V1$
- $N1$ `fca:hasIntent` $V2$
- $N2$ `fca:hasIntent` $V1$
- NOT EXISTS {$N2$ `fca:hasIntent` $V2$ }
- $N1$ `fca:isAscendantOf` $N2$

Ces critères étant identifiés au format SPARQL, ne reste plus alors qu'à les intégrer dans la requête de type CONSTRUCT de la figure 4.4 pour obtenir un graphe référençant l'ensemble de ces relations sous la forme de triplets $?V2$ fca : $partSpecial$ $?V1$. Le graphe calculé a alors pu simplement être ajouté à la base de données afin de laisser l'information disponible.

L'intérêt de cette relation est qu'elle permet d'identifier dans un premier temps les cas de variables pour lesquelles le code contrôlé est inclus strictement dans le code contrôlé par une autre variable. Il suffit de rechercher les cas où les deux variables n'ont une relation de spécialisation que dans un sens. Cette relation permet également d'identifier les cas pour lesquels les imbrications de contrôle ne sont pas systématiques. Plus de détails sur l'intérêt de ces relations d'imbrication sont fournis dans le chapitre suivant.

```
CONSTRUCT {?V1 fca:partSpecial ?V2}
WHERE{

N1 fca:hasIntent V1
N1 fca:hasIntent V2
N2 fca:hasIntent V1
 NOT EXISTS \{N2 fca:hasIntent V2 \}
N1 fca:isAscendantOf N2
FILTER regex(str(?control), ‘‘control/’’)
}
```

Figure 4.4 Requête SPARQL pour définir la spécialisation partielle

4.6.2 Possibilités futures

Au delà de l'exemple d'inférences proposé précédemment, il est important de voir que les possibilités pour ce type de démarche sont extrêmement larges. En effet, les données disponibles et le choix d'implantation permettent une grande liberté.

Par exemple, si l'on considère l'optique d'extraction de lignes de produit, il est tout à fait envisageable d'étendre automatiquement le modèle de fonctionnalités de la future version en associant les variables de configuration, ou les lignes de code, ou même les composants existant aux futurs composants via des inférences à définir par des experts.

Dans un autre ordre d'idée, si l'on considère le système dans un objectif d'amélioration du code et de rassemblement de fonctionnalités pour une meilleure compréhension, la construction d'inférences pour marquer le code devant être rassemblé ou, au contraire le code devant être ignoré dans le processus est tout à fait envisageable.

Toutefois, les limites de ces approches se situent dans la définition des inférences, en effet, il n'est pas possible de construire ce type de règles automatiques sans posséder une connaissance approfondie du système. C'est là que la création de la base de connaissances prends tout son sens, permettant de mettre à disposition des experts du logiciel l'ensemble des résultats des présents travaux, et leur permettant ainsi de faire leur propres observations et de rechercher leurs propres propriétés.

En conclusion, nous avons vu dans ce chapitre les contraintes imposées pour l'établissement d'une base de connaissances utilisant les résultats des présents travaux, nous avons également présenté la solution et son implantation accompagnée d'exemples d'utilisation mettant en évidence les possibilités de la solution choisie. La base de données implantée est désormais déployée chez l'entreprise partenaire et fonctionnelle.

CHAPITRE 5

Étude des cas de partage de contrôle entre variables

L'un des objectifs des travaux présentés dans le présent mémoire est de faciliter la ré-ingénierie du logiciel FMS. L'une des difficultés posées pour effectuer cette refactorisation concerne les partages de contrôle entre variables. Dans ce chapitre, nous présentons le problème et présentons les différentes occurrences rencontrées ainsi qu'une analyse de ces différents cas.

5.1 Description du problème

Comme nous avons pu le voir dans le chapitre 3, il y a de nombreux recoupements entre les contrôles des variables de configuration au niveau du code au travers des relations *Associated* et *Complementary*. Ces recoupements peuvent prendre différentes formes qui sont décrites dans la section suivante.

Le principal problème que posent ces partages de contrôle est qu'il n'est pas nécessairement simple d'identifier le type de partage dont il s'agit et que, si certains de ces partages de contrôle sont voulus, d'autres peuvent résulter de mauvaises habitudes de programmation ou de mauvais choix de conception.

De plus, ces partages de contrôle peuvent amener de nombreuses difficultés dans une démarche d'évolution ou d'extraction de lignes de produit. Par exemple, si deux variables correspondant à des produits totalement différents contrôlent la même portion de code, il sera nécessaire de rechercher la meilleure solution pour l'extraction des lignes de produits ; il pourrait être plus intéressant de créer des clones de code pour séparer clairement les contrôles, ou au contraire de conserver le code partagé tel quel dans un module. La suggestion des clones provient du fait que, dans le contexte avionique, on souhaite privilégier la présence de code exécuté, potentiellement en plusieurs exemplaires, à la présence de code exécuté seulement sur certains produits. En effet, ce dernier doit être testé même pour les produits ne l'exécutant pas, ce qui cause de forts surcoûts en termes de tests et de certification. Il faut néanmoins reconnaitre les limites de l'approche par ligne de produits qui, malgré ses avantages, possède également certains inconvénients.Ces choix pourraient être faits lors de la ré-ingénierie. Cependant, à ce moment là, il pourrait s'avérer difficile de conserver une cohérence dans les décisions prises. En effet, une succession de décisions prises au dernier moment et sans comparaison préalable pourrait amener des incohérence ou des contradictions. De plus, la

systématisation de ces décisions amènerait nécessairement la question de l'identification des différents cas possibles.

Il est donc important d'identifier les différentes possibilités de partage de contrôle entre variables mais aussi de déterminer si elles résultent d'une volonté claire d'interdépendance ou au contraire d'une implantation qui pourrait être simplifiée ou optimisée.

5.2 Cas de partage de contrôle entre variables

Dans cette section, nous présentons les différentes possibilités de partage de contrôle au sein du système, nous définissons des critères d'identification à partir des résultats de nos analyses et nous étudions la distribution des différents cas au sein du système.

5.2.1 Types de partages

Dans le cadre d'un partage de contrôle entre deux variables, plusieurs cas sont possibles et ces cas dépendent de l'échelle à laquelle on se situe pour effectuer l'analyse. Bien que les cas de partages identifiés ici soient considérés au niveau du système, plusieurs des critères développés se placent à l'échelle du fichier de code source. De plus, dans le cadre des travaux de [23] sur lesquels se fondent les présents travaux, l'analyse effectuée est inter-procédurale, les partages de contrôle identifiés ne sont donc pas nécessairement uniquement syntaxiques. Dans la mesure où pour une ré-ingénierie, les partages de contrôle les plus gênants sont ceux qui activent l'exécution de code, nous nous intéressons donc ici aux cas du code contrôlé avec le motif GAIN uniquement. Les motifs de contrôle LOSS, SOMEHOWPLUS et SOMEHOWMINUS sont également importants mais, en raison de la proportion de code très inférieure qu'ils impliquent, leur traitement est moins prioritaire pour l'industrie.

Variables incluses Le premier cas envisageable lors d'un partage de contrôle consiste en une inclusion totale du code contrôlé par une variable dans le code contrôlé par une autre. L'expression formelle de cette relation avec $Var1$ incluse dans $Var2$ est alors :

$$(GAIN(Var1) \subsetneq GAIN(Var2)) \tag{5.1}$$

$Var1$ et $Var2$ sont deux variables et $GAIN(Var)$ l'ensemble des lignes contrôlées par Var avec le motif GAIN. Ce cas de figure peut présenter deux types de structure dans le code : ou bien $Var1$ est toujours testée au sein du code contrôlé par $Var2$, ou bien les conditions d'exécution du code pour $Var1$ testent toujours $Var2$ en même temps sans que la réciproque soit vraie.

Si l'on s'appuie sur la définition de la spécialisation partielle de la section 4.6.1, on peut remarquer que la relation d'inclusion correspond au cas où la spécialisation partielle est à sens unique entre les deux variables. En effet, si tout le code contrôlé par une variables avec le motif GAIN est strictement inclus dans celui contrôlé par une autre avec le même motif, alors la définition même de l'analyse de concepts formels impose que les deux variables aient une relation de spécialisation partielle et que cette relation ne soit pas réciproque.

La requête SPARQL pour obtenir la liste des couples de variables incluses telle que la seconde variable du couple est incluse dans la première est la suivante :

```
SELECT DISTINCT ?var1 ?var2 WHERE{
?var1 fca:partialSpecial ?var2.
FILTER NOT EXISTS {?var2 fca:partialSpecial ?var1.}
}
```

Le partage de contrôle par inclusion de variables peut permettre une hiérarchisation des variables dans le sens où la variable qui est incluse n'est utilisée que dans le contexte où l'autre variable est active.

Inclusion mutuelle Le partage de contrôle par inclusion mutuelle est défini à partir de l'analyse du partage de contrôle dans chaque fichier. En effet, il est possible que dans un fichier le code contrôlé par une variable $Var1$ soit inclus dans le code contrôlé par $Var2$, mais il est également possible que cette situation soit inversée dans un autre fichier sans pour autant que les deux variables interagissent dans tous les fichiers où elles sont présentes. C'est ce cas de figure que nous appelons l'inclusion mutuelle. Une telle relation s'exprime alors de la manière suivante :

$$\exists f1, f2 \in F | (Var1(f1) \subsetneq Var2(f1)) AND (Var2(f2) \subsetneq Var1(f2)) \tag{5.2}$$

$Var1$ et $Var2$ sont deux variables, F l'ensemble des fichiers et $Var(f)$ l'ensemble des lignes contrôlées par Var dans $f \in F$.

La recherche des cas d'inclusion mutuelle est plus complexe à faire à l'aide d'une seule requête SPARQL dans la base de données RDF. Dans un premier temps nous identifions donc les cas pour lesquels la relation de spécialisation partielle définie précédemment est réciproque, en effet, le partage de contrôle par inclusion mutuelle n'est possible que si la spécialisation partielle est à double sens, bien que cela ne soit pas suffisant (cela caractérise simplement une intersection dans le code contrôlé sans inclusion totale). Nous nous sommes ensuite servi de l'interface de programmation de Sesame en Java pour implanter l'algorithme

5.1 qui nous a permis d'identifier les couples de variables à inclusions mutuelles.

Les ensembles définis aux lignes 1, 2, 3 et 9 de la figure 5.1 correspondent au résultat d'une requête SPARQL dans la base de données RDF. De plus, quelques modifications simples de cet algorithme permettent d'obtenir, pour chaque couple considéré, la liste des lignes de code communes pour de plus amples vérifications.

Contrôle simultané Le cas du contrôle simultané est apparenté au cas de l'inclusion mutuelle dans le sens où l'on s'intéresse à nouveau à des inclusions réciproques. Cependant, la relation de contrôle simultané consiste à l'égalité des ensembles de code contrôlé. C'est à dire que, quel que soit le fichier, les deux variables sont toujours testées dans la même condition. C'est à dire, avec les notations précédemment utilisées, que deux variables $Var1$ et $Var2$ sont à contrôle simultané si :

$$\nexists f1 \in F | Var1(f1) \neq Var2(f1) \tag{5.3}$$

L'identification des couples de variables à contrôle simultané peut également se faire à l'aide du treillis de FCA entre les variables de configuration et le code. En effet, si les deux variables contrôlent exactement le même code, elles appartiendront au même concept dans le treillis, mais, puisqu'elles ne possèdent aucune différence dans leur code contrôlé, il n'existe aucune relation de spécialisation partielle entre les deux, cette relation caractérisant le partage de code avec des différences.

Le code suivant présente la requête SPARQL à exécuter pour identifier les couples de variables à contrôle simultané.

```
SELECT DISTINCT ?var1 ?var2 WHERE{
?node fca:hasIntent/VAR_CODE ?var1.
?node fca:hasIntent/VAR_CODE ?var2.
FILTER EXISTS {?node fca:hasextent/VAR_CODE ?line.}
FILTER NOT EXISTS {?var1 fca:partialSpecial ?var2.}
FILTER NOT EXISTS {?var2 fca:partialSpecial ?var1.}
FILTER(str(?var1) < str(?var2))
}
```

Intersection de contrôle L'intersection de contrôle correspond à tous les cas ne remplissant pas les conditions des cas précédents. C'est à dire que les deux variables partagent effectivement du code, mais de manière trop désordonnée pour être véritablement caractérisée.

```
1  couples <- {couples de variables a specialisation partielle a double sens}
2  variables <- {variables dans couples}
3  fichiers <-{fichiers du systeme}
4  controle <- initialisation a vide
5  resultats <- [ ]
6
7  Pour var dans variables:
8          Pour file dans fichiers:
9                  controle[var][file]<- {lignes controlees par var dans file}
10
11 Pour cpl dans couples:
12         inclusionDIR<- false
13         inclusionINDIR<- false
14         Pour file dans fichiers:
15                 Si (controle[cpl[0]][file]IN controle[cpl[1]][file])
16                         inclusionDIR<-true
17                 Sinon Si (controle[cpl[1]][file] IN controle[cpl[0]][file])
18                         inclusionINDIR<-true
19
20
21         Si (inclusionDIR && inclusionINDIR)
22                 resultats.add(cpl)
```

Figure 5.1 Algorithme de recherche des variables à inclusion mutuelle

Les cas d'intersection de contrôle sont obtenus à partir des couples de variables vérifiant une spécialisation partielle à double sens mais sans inclusion mutuelle. Il suffit donc de filtrer les cas d'inclusions mutuelles dans la liste déjà obtenue.

5.2.2 Distributions

Nous présentons dans cette sous-section les résultats de l'analyse des partages de contrôle. La table 5.1 présente le nombre de couples de variables trouvés pour chacun des types de partage de contrôle. Nous nous intéressons ensuite à la proportion de code impliquée dans les différentes possibilités de contrôle de code. Étant données les définitions des partages de contrôle, les partages sont exclusifs les uns des autres.

En utilisant les requêtes SPARQL définies précédemment, nous sommes en mesure d'obtenir très rapidement les résultats. C'est ainsi que nous trouvons 24 couples où le contrôle d'une variable est inclus dans celui d'une autre et 16 couples de variables contrôlant exactement le même code.

Nous trouvons également 39 couples de variables pour lesquelles les inclusion de contrôle ne se font pas dans le même ordre dans tous les fichier.

Et enfin, le système comporte 88 couples de variables partageant un contrôle sur du code au sein du logiciel sans pour autant répondre à l'un des cas précédent.

La figure 5.2 présente, au sein du code contrôlé avec le motif GAIN, la proportion de code impliqué dans chacun des partages de contrôle identifiés. On peut tout d'abord constater que, sur les près de 43 000 lignes de code contrôlées avec le motif GAIN, une très large majorité (76%) n'est contrôlée que par une seule variable à la fois.

On peut également constater que le nombre de lignes de codes contrôlées par des variables incluses est extrêmement faible, seulement 349 lignes impliquées et les lignes de code contrôlées par des variables à contrôle simultané représentent seulement 3% de l'ensemble.

Enfin, les variables à inclusion mutuelles et celles à intersection de contrôle représentent, combinées, environ 8500 lignes de code, soit moins de 2% du logiciel bien que cela représente 20% du code contrôlé avec le motif GAIN.

Tableau 5.1 Nombre de couples de variables partageant un contrôle de code

Types de partage	**Nombre d'instances**
Variables incluses	24
Inclusion mutuelle	39
Contrôle simultané	16
Intersection de contrôle	88

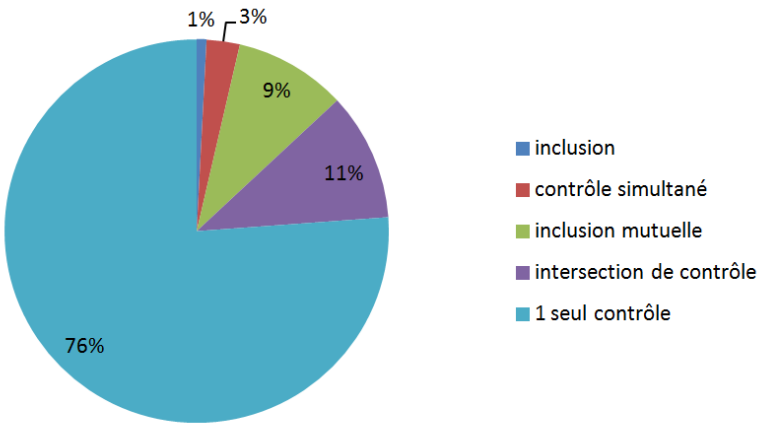

Figure 5.2 Proportion de lignes de code contrôlées avec GAIN impliquées dans chaque partage de contrôle

5.3 Discussion

Après avoir présenté les types de partage de contrôle que nous identifions et les résultats obtenus sur le FMS, nous discutons ici des résultats et de l'interprétation qui peut en être faite.

Dans un premier temps, nous pouvons nous intéresser aux variables incluses. En effet, le partage de contrôle par cette relation peut laisser penser que la variable dont le code est inclus est une particularisation de l'autre. Dans le contexte d'une ré-ingénierie par ligne de produits, cette information est particulièrement intéressante. En effet, il serait possible d'extraire la hiérarchie de ces variables pour construire le modèle de lignes de produit. Nous avons pu vérifier avec nos partenaires expert que c'est effectivement le cas. Malheureusement, le petit nombre de ces couples de variables et de lignes de code montre que c'est une portion très faible du logiciel qui est concernée.

Les variables à contrôle simultané posent un problème tout autre. En effet, dans le contexte qui est le notre, avec uniquement les contrôle de type GAIN pris en compte, les variables à contrôle simultané exécutent donc exactement le même code. Dans ce contexte, il semblerait pertinent de fusionner les deux variables puisque l'utilisation de l'une ou de l'autre des variables ne changerait rien au comportement du système. De même, pour une ré-ingénierie par ligne de produits, la redondance de variables pourrait poser des difficultés d'identification de fonctionnalités. Toutefois, le faible nombre d'instances de ce type trouvées dans le système, seulement 16, laisse penser que la correction de ces partages de contrôle pourrait être faite rapidement. Le nombre de lignes de code associées à ce motif, environ 1000 confirme cette

idée.

Nous pouvons ensuite nous intéresser aux variables à inclusion mutuelle. Ces couples de variables sont intéressants à identifier car, dans le contexte du domaine avionique dans lequel on se place, ce type de chevauchement ne devrait pas être répandu. En effet, d'après nos interlocuteurs chez l'entreprise partenaire, lorsque les contrôles sont imbriqués, ce devrait toujours être dans le même sens, sauf pour quelques exceptions voulues par l'architecture du système. Les doubles inclusions résultent donc probablement de mauvaises habitudes d'implémentation qui devraient être refactorisées pour aboutir à un système plus simple et plus clair. La définition de stratégie à adopter pour une telle refactorisation et son coût restent encore à définir. Nous avons présenté la liste des couples présentant des inclusions doubles et le retour des experts du système a permis d'éliminer une grande partie de ces couples car, même si l'analyse statique détecte certains chemins d'exécution, les contraintes de programmation et de configuration rendent ce chemin impossible dynamiquement, l'activation simultanée de certaines variables étant impossible à cause d'une relation sémantique négligée au cours de l'extraction des contrôles. Après avoir filtré ces combinaisons impossibles vérifiant les conditions au sein du treillis et du code mais incompatibles du fait de leur rôle dans le système, nous conservons 17 couples à inclusions mutuelles nécessitant donc une refactorisation. Encore une fois, le nombre est assez faible pour envisager un traitement manuel, ce qui est confirmé par le nombre de lignes de code considéré, environ 4000, qui n'est pas négligeable mais est loin d'être rédhibitoire pour ce type de traitements.

Le nombre d'intersections de contrôle que nous avons pu identifier au sein du système fait de ce phénomène un aspect non négligeable du logiciel. C'est également une source potentielle de difficultés pour les opérations de maintenance et de ré-ingénierie. En effet, lorsque deux variables interagissent au niveau du code sans vérifier les autres cas, cela veut dire que les variables ne sont, a priori, pas liées en termes de fonctionnalités. Dans ce cas, il semblerait important de trouver une solution de refactorisation permettant de découpler les variables. Sans ce type de correctif, l'extraction d'une ligne de produits pour peupler un feature model s'avèrerait extrêmement complexe car l'intersection de contrôle brise la structure arborescente de ce type de modèles. Des pistes pouvant être envisagées seraient la création de clones de code ou bien la création de nouvelles variables. Le nombre de lignes de code impliquées dans ce partage de contrôle étant d'environ 4500 et étant donné la complexité du système, nous estimons qu'un traitement au cas par cas de ces partages de contrôle serait plus pertinent et possible.

Au travers de l'étude des partages de contrôle, nous avons pu identifier quatre relations plus précises que la relation Associated définie au chapitre 3. Toutefois, cette approche présente plusieurs limites, notamment au niveau des différents motifs de contrôle. Il serait né-

cessaire de mener la même étude sur les autres motifs de contrôle pour que les présentes conclusions puissent être appliquées. De plus, les interactions combinées sur les différents motifs sont probablement plus complexes que celles identifiées ici. L'identification des motifs de contrôle que nous proposons est cependant généralisable, quoique l'interprétation qui en est faite doive changer en fonction des motifs. De plus, étant donné l'objectif sous-jacent de ré-ingénierie par ligne de produit, l'étude des partages de contrôle sur le motif GAIN, représentant on le rappelle plus de 90% du code contrôlé, et donc le plus à même d'être réutilisé, porte un poids tout à fait significatif dans l'analyse finale comparativement aux autres motifs.

Dans ce chapitre, nous avons pu constater que les partages de contrôle entre variables peuvent poser des difficultés pour l'évolution ou la compréhension du système et que ces partages de contrôle peuvent prendre différentes formes. Nous avons également défini des critères permettant d'identifier ces différentes formes à partir de la base de données définie au chapitre précédent et plus particulièrement du treillis de FCA entre les variables de configuration et le code contrôlé avec le motif GAIN. Nous avons également présenté les résultats de cette analyse sur notre cas d'étude en termes de couples de variables et de code impliqué et discuté de l'interprétation que nous en faisons ainsi que des limites de cette approche. En conclusion, les observations que nous avons faites suggèrent que des actions soient définies pour réduire la complexité du système et découpler les variables afin de simplifier une extraction de ligne de produits, mais des analyses plus poussées sur les autres motifs de contrôle pourraient aider à définir certaines de ces actions.

CHAPITRE 6

CONCLUSION

6.1 Synthèse des travaux

Les travaux présentés dans ce mémoire atteignent les objectifs de recherche proposés initialement, c'est-à-dire la définition d'une classification des variables, l'analyse des partages de contrôle entre variables, la définition d'une base de connaissances pour conserver et utiliser les résultats d'analyse et l'application de ces solutions dans le cas de l'entreprise partenaire. L'analyse de concepts formels et la définition de nouvelles métriques nous ont permis de construire une classification des variables de configuration et des principales relations qu'elles peuvent présenter avec d'autres variables. Nous avons également appliqué cette classification aux variables de configuration du FMS et les résultats que nous avons obtenus ont pu être validés par un expert du système.

Nous avons ensuite défini une nouvelle ontologie permettant de représenter les connaissances acquises sur le logiciel et construit une base de données permettant de conserver et d'interroger ces données. L'implantation de cette base de données a également impliqué la conception d'une interface en Java, implémentant des requêtes semblant plus intéressantes et pouvant être récurrentes pour les utilisateurs au sein de l'entreprise.

Enfin, à partir de la base de données et de l'analyse de concepts formels, nous avons pu identifier différents types de partage de contrôle de code entre variables et avons pu étudier leur distribution au sein du logiciel.

6.2 Limitations de la solution proposée

Bien que les objectifs de recherche ont été atteints, les analyses présentées dans ce mémoire présentent certaines limites. Tout d'abord dans la définition de la classification, nous nous appuyons sur des seuils statistiques au sein du code pour confirmer l'appartenance à une catégorie, ces seuils statistiques ont fourni des résultats évalués positivement dans le cas du FMS, mais l'utilisation sur un autre système pourrait nécessiter un ajustement de ces seuils, d'autres études sur d'autres systèmes seraient donc nécessaires.

Par ailleurs, l'ontologie que nous avons définie semble pertinente et utilisable dans le cadre du présent projet. Toutefois, cela n'est qu'une représentation des résultats des analyses effectuées. Il n'est pas du tout certain que la même ontologie puisse s'avérer pertinente sur un projet différent dont les ressources diffèrent.

Enfin, notre analyse des partages de contrôle se fonde uniquement sur le code contrôlé par le motif GAIN et cela représente une restriction dans l'analyse des interactions de variables. En raison de la proportion de code contrôlé par ce motif sur l'ensemble des lignes contrôlées, nous estimons que notre analyse est une bonne approximation des résultats, toutefois, des études supplémentaires s'avèrent nécessaire pour le confirmer.

6.3 Améliorations futures

Les limitations présentées dans la section précédente ne sont pas sans solutions. Les travaux futurs que nous envisageons se situent donc à plusieurs niveaux. Nous estimons tout d'abord que l'application de nos analyses à d'autres systèmes configurés dynamiquement serait une progression intéressante permettant de généraliser l'utilisation de la classification présentée. De plus, il nous semble important pour les travaux futurs de prévoir de compléter l'analyse des partages de contrôle en tenant compte des différents motifs.

RÉFÉRENCES

[1] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," Tech. Rep., 2001.

[2] R. Kneuper, in *Software Management*, ser. LNI, T. Spitta, J. Borchers, and H. M. Sneed, Eds. GI, pp. 171–172.

[3] 2007. [Online]. Available : http://www.faa.gov/news/fact_sheets/news_story.cfm?newsid=8145

[4] A. MacDonald, D. Russell, and B. Atchison, "Model-driven development within a legacy system : an industry experience report," in *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, March 2005, pp. 14–22.

[5] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering : Foundations, Principles and Techniques.* Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005.

[6] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," November 1990.

[7] 2009. [Online]. Available : http://en.wikipedia.org/wiki/File:E-shopFM.jpg/

[8] B. Ganter and R. Wille, *Formal Concept Analysis : Mathematical Foundations.* Springer Verlag, 1999.

[9] 2006. [Online]. Available : http://en.wikipedia.org/wiki/Formal_concept_analysis#mediaviewer/File:Concept_lattice.svg

[10] H. Gomaa, "Designing software product lines with uml," in *Software Engineering Workshop - Tutorial Notes, 2005. 29th Annual IEEE/NASA*, 2005, pp. 160–216.

[11] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch, "Modeling dependencies in product families with covamof," in *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, 2006, pp. 9 pp.–307.

[12] K. Bak, "Modeling and analysis of software product line variability in clafer," Master's thesis, University of Waterloo, 11/2013 2013. [Online]. Available : http://hdl.handle.net/10012/8039

[13] "Feature and class models in clafer : Mixed, specialized, and coupled," 2010.

[14] K. Czarnecki and et al., "Cardinality-based feature modeling and constraints : A progress report," 2005.

[15] O. Haugen, A. Wąsowski, and K. Czarnecki, "Cvl : Common variability language," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC '13. New York, NY, USA : ACM, 2013, pp. 277–277. [Online]. Available : http://doi.acm.org/10.1145/2491627.2493899

[16] K. Czarnecki and A. Wasowski, "Feature diagrams and logics : There and back again," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Sept., pp. 23–34.

[17] M. Becker, "Towards a general model of variability in product families," in *in Proceedings of the First Workshop on Software Variability Management*, 2003.

[18] M. Eriksson, J. Börstler, and K. Borg, "The pluss approach : Domain modeling with features, use cases and use case realizations," in *Proceedings of the 9th International Conference on Software Product Lines*, ser. SPLC'05, 2005, pp. 33–44.

[19] A. van der Hoek, "Design-time product line architectures for any-time variability." *Sci. Comput. Program.*, vol. 53, no. 3, pp. 285–304, 2004.

[20] L. A. Zaid, F. Kleinermann, and O. De Troyer, "Feature assembly : A new feature modeling technique," in *Proceedings of the 29th International Conference on Conceptual Modeling*, ser. ER'10. Berlin, Heidelberg : Springer-Verlag, 2010, pp. 233–246. [Online]. Available : http://dl.acm.org/citation.cfm?id=1929757.1929780

[21] L. Hotz, K. Wolter, T. Krebs, J. Nijhuis, M. Sinnema, S. Deelstra, and J. MacGregor, *Configuration in Industrial Product Families - The ConIPF Methodology.* AKA-Verlag, 2006.

[22] C. W. Krueger, "Variation management for software production lines," in *Proceedings of the Second International Conference on Software Product Lines*, ser. SPLC 2. London, UK, UK : Springer-Verlag, 2002, pp. 37–48. [Online]. Available : http://dl.acm.org/citation.cfm?id=645882.672255

[23] M. Ouellet, "Localisation de fonctionnalites par analyse statique dans du code avionique configure dynamiquement," Master's thesis, École Polytechnique de Montréal, Canada, 2012.

[24] C. Stoermer and L. O'Brien, "Map - mining architectures for product line evaluations," in *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, 2001, pp. 35–44.

[25] L. O'Brien, F. Hansen, R. Seacord, and D. Smith, "Mining and managing software assets," in *Software Technology and Engineering Practice, 2002. STEP 2002. Proceedings. 10th International Workshop on*, Oct 2002, pp. 82–90.

[26] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba, "Flip : Managing software product line extraction and reaction with aspects," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, 2008, pp. 354–354.

[27] M. Couto, M. Valente, and E. Figueiredo, "Extracting software product lines : A case study using conditional compilation," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, 2011, pp. 191–200.

[28] B. Zhang, "Extraction and improvement of conditionally compiled product line code," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 2012, pp. 257–258.

[29] M. Valente, V. Borges, and L. Passos, "A semi-automatic approach for extracting software product lines," *Software Engineering, IEEE Transactions on*, vol. 38, no. 4, pp. 737–754, 2012.

[30] Y. Xue, "Reengineering legacy software products into software product line based on automatic variability analysis," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 1114–1117.

[31] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May, pp. 461–470.

[32] R. Stoiber and M. Glinz, "Feature unweaving : Efficient variability extraction and specification for emerging software product lines," in *Software Product Management (IWSPM), 2010 Fourth International Workshop on*, 2010, pp. 53–62.

[33] P. Frenzel, R. Koschke, A. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, Oct 2007, pp. 160–169.

[34] G. Murphy and D. Notkin, "Reengineering with reflexion models : a case study," *Computer*, vol. 30, no. 8, pp. 29–36, Aug 1997.

[35] A. Harhurin and J. Hartmann, "Service-oriented commonality analysis across existing systems," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, Sept 2008, pp. 255–264.

[36] T. Tilley, R. Cole, P. Becker, and P. Eklund, "A survey of formal concept analysis support for software engineering activities," in *Formal Concept Analysis : Foundations and Applications*, ser. Lecture Notes in Computer Science, B. Ganter, G. Stumme, and R. Wille, Eds. Berlin/Heidelberg : Springer, 2005, vol. 3626, pp. 250–271.

[37] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining security-sensitive operations in legacy code using concept analysis," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, May 2007, pp. 458–467.

[38] H. Eyal-Salman, A.-D. Seriai, and C. Dony, "Feature-to-code traceability in a collection of software variants : Combining formal concept analysis and information retrieval," in *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, Aug 2013, pp. 209–216.

[39] Y. Yang, X. Peng, and W. Zhao, "Domain feature model recovery from multiple applications using data access semantics and formal concept analysis," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, 2009, pp. 215–224.

[40] S. Yang and P. Xiao-Zhong, "Recovering crosscutting concern from legacy software based on use-cases driven formal concept analysis," in *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, Dec 2010, pp. 1–4.

[41] H. Kazato, S. Hayashi, S. Okada, S. Miyata, T. Hoshino, and M. Saeki, "Feature location for multi-layer system based on formal concept analysis," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, March 2012, pp. 429–434.

[42] T. Satyananda, D. Lee, S. Kang, and S. Hashmi, "Identifying traceability between feature model and software architecture in software product line using formal concept analysis," in *Computational Science and its Applications, 2007. ICCSA 2007. International Conference on*, 2007, pp. 380–388.

[43] F. Loesch and E. Ploedereder, "Restructuring variability in software product lines using concept analysis of product configurations," in *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, 2007, pp. 159–170.

[44] ——, "Optimization of variability in software product lines," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Sept 2007, pp. 151–162.

[45] Y. Zhao, J. Dong, and T. Peng, "Ontology classification for semantic-web-based software engineering," *Services Computing, IEEE Transactions on*, vol. 2, no. 4, pp. 303–317, Oct 2009.

[46] J. Zhai, W. Liu, Y. Liang, and J. Jiang, "Fuzzy knowledge representation for fuzzy systems based on ontology and rdf on the semantic web," in *Information and Automation, 2008. ICIA 2008. International Conference on*, June 2008, pp. 1101–1105.

[47] R. Witte, Y. Zhang, and J. Rilling, "Empowering software maintainers with semantic web technologies," in *Proceedings of the 4th European Conference on The Semantic Web : Research and Applications*, ser. ESWC '07, 2007, pp. 37–52.

[48] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan, "A semantic web approach to feature modeling and verification," in *In Workshop on Semantic Web Enabled Software Engineering (SWESE'05*, 2005.

[49] J. Wang and K. He, "Towards representing fca-based ontologies in semantic web rule language," in *Computer and Information Technology, 2006. CIT '06. The Sixth IEEE International Conference on*, Sept 2006, pp. 41–41.

[50] T. Parr and R. W. Quong, "Adding semantic and syntactic predicates to ll(k) : pred-ll(k)," in *CC'94 : Proceedings of the fifth international conference on Compiler Construction*. Springer Verlag, 1994, pp. 263–277.

[51] J.-L. Guigues and V. Duquenne, "Familles minimales d'implications informatives résultant d'un tableau de données binaires," *Mathématiques et Sciences Humaines*, vol. 95, pp. 5–18, 1986.

[52] 1999. [Online]. Available : http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[53] 2014. [Online]. Available : http://www.w3.org/TR/rdf-syntax-grammar/

[54] 2004. [Online]. Available : http://franz.com/agraph/allegrograph/

[55] 2007. [Online]. Available : http://4store.org/

[56] 2004. [Online]. Available : http://openrdf.org